HIERARCHICAL REINFORCEMENT LEARNING IN CONTINUOUS STATE AND MULTI-AGENT ENVIRONMENTS

A Dissertation Presented

by

MOHAMMAD GHAVAMZADEH

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2005

Computer Science

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **SEP 2005**	2. REPORT TYPE	3. DATES COVERED **-**
4. TITLE AND SUBTITLE **Hierarchical Reinforcement in Continuous State and Multi-Agent Environments**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Defense Advanced Research Projects Agency,3701 North Fairfax Drive,Arlington,VA,22203-1714**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR’S ACRONYM(S)
		11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**173**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

HIERARCHICAL REINFORCEMENT LEARNING IN CONTINUOUS STATE AND MULTI-AGENT ENVIRONMENTS

A Dissertation Presented

by

MOHAMMAD GHAVAMZADEH

Approved as to style and content by:

Sridhar Mahadevan, Chair

Andrew G. Barto, Member

Victor R. Lesser, Member

Weibo Gong, Member

W. Bruce Croft, Department Chair
Computer Science

To my parents.

ACKNOWLEDGMENTS

I must begin by thanking my mother and then proceed to ask her to forgive me for yet another failing: I am absolutely incapable of expressing the depth of my gratitude for her endless love, support, and encouragement.

I am deeply grateful to my advisor Sridhar Mahadevan, whose guidance, support, and patience were instrumental in bringing this work to fruition. Sridhar gave me tremendous freedom to explore and try new ideas, which has had an essential role in my growth as a researcher. Thank you Sridhar.

During my graduate studies at UMass I have had the opportunity to collaborate with Andy Barto. I have found Andy an outstanding and visionary researcher, and a wonderful human being. It was a great honor and a real pleasure for me to have him as a member of my thesis committee.

I am also indebted to the other members of my committee for their patience in reading drafts of my thesis, their insightful comments, and their stimulating questions during my defense. I thank Victor Lesser for his constant support, and for helping me better understand research directions in multi-agent systems; and Weibo Gong for inspiring conversations.

I must thank Doina Precup heartily for her unwavering support while a long visa delay had interrupted my research and almost every other aspect of my life. It is amazing how one's career and dignity can fall at the mercy of such a seemingly banal uncertainty as a visa delay. I am indebted for her support at such a time: she made every effort to make me feel part of the community at the computer science department at McGill university.

Many others have shared their insights and contributed to the development of the ideas in the thesis. I especially thank Balaraman Ravindran and my old buddy Khashayar Rohan-

imanesh for many useful conversations and more important for their precious friendship. I thank Andy Fagg and Mike Rosenstein for exposing me to a wide variety of topics in continuous state and action reinforcement learning. I never forget Andy's friendship, his down-to-earth manner, and his tasty and fresh salsas. I thank Mike who made organizing a workshop at AAAI-2004 a joyful and educational experience for me.

I want to thank Caro Locus and Ali M. Eydgahi, my M.S. and B.S. advisors from University of Tehran, Iran. They taught me how to be a researcher, how to better express my ideas, and helped me in writing my first research papers. I also want to thank Abdol Esfahanian without whom it would not have been possible for me to pursue my education in the United States of America.

I would like to thank all the members of the Autonomous Learning Laboratory at UMass, past and present, for their friendship, for their constant support and encouragement, for giving useful feedback during my practice talks and lab-meeting presentations, and finally for taking care of my cubicle during my unwanted one-year absence. Thank you Colin Barringer, Jad Davis, Andy Fagg, Jeffrey Johns, Anders Jonsson, George Konidaris, Victoria Manfredi, Amy McGovern, Sarah Osentoski, Ted Perkins, Marc Pickett, Balaraman Ravindran, Khashayar Rohanimanesh, Mike Rosenstein, Suchi Saria, Ashvin Shah, Özgür Şimşek, Andrew Stout, Chris Vigorito, and Pippin Wolfe for making our lab such an excellent and enjoyable environment for research.

I am also grateful to the members of our small Autonomous Agents Laboratory at Michigan State University, with whom I learned about new research directions, open problems, and solution techniques in Artificial Intelligence, Machine Learning, and Reinforcement Learning: Natalia Hernandez Gardiol, Rajbala Makar, Silviu Minut, Khashayar Rohanimanesh, and Georgios Theocharous.

I am proud to belong to an intellectual community that treats hopeful, young graduate students with the same respect as senior researchers. Some of the members of this community who have been particularly helpful and kind to me, and their useful comments

contributed to the quality of this document are David Andre, Bernhard Hengst, Shie Mannor, Doina Precup, Richard Sutton, and Prasad Tadepalli.

The material in this work is based upon work carried out in the Autonomous Agents Laboratory in the Department of Computer Science and Engineering at Michigan State University, under the DARPA contract DAANO2-98-C-4025, and the Autonomous Learning Laboratory in the Department of Computer Science at University of Massachusetts Amherst, under the NASA contract NAg-1445 #1, and the NSF grant ECS-0218125.

ABSTRACT

HIERARCHICAL REINFORCEMENT LEARNING IN CONTINUOUS STATE AND MULTI-AGENT ENVIRONMENTS

SEPTEMBER 2005

MOHAMMAD GHAVAMZADEH

B.Sc., UNIVERSITY OF TEHRAN, IRAN

M.Sc., UNIVERSITY OF TEHRAN, IRAN

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Sridhar Mahadevan

This dissertation investigates the use of hierarchy and abstraction as a means of solving complex sequential decision making problems such as those with continuous state and/or continuous action spaces, and domains with multiple cooperative agents. This thesis develops several novel extensions to hierarchical reinforcement learning (HRL), and designs algorithms that are appropriate for such problems.

It has been shown that the **average reward** optimality criterion is more natural than the more commonly used discounted criterion for *continuing* tasks. This thesis investigates two formulations of HRL based on the *average reward* semi-Markov decision process (SMDP) model, both for discrete-time and continuous-time. These formulations correspond to two notions of optimality that have been explored in previous work on HRL: *hierarchical optimality* and *recursive optimality*. Novel discrete-time and continuous-time algorithms,

termed **hierarchically optimal average reward RL** (HAR) and **recursively optimal average reward RL** (RAR) are presented, which learn to find hierarchically and recursively optimal average reward policies. Two automated guided vehicle (AGV) scheduling problems are used as experimental testbeds to empirically study the performance of the proposed algorithms.

Policy gradient reinforcement learning (PGRL) methods have several advantages over the more traditional value function RL algorithms in solving problems with continuous state spaces. However, they suffer from slow convergence. This thesis defines a family of **hierarchical policy gradient RL** (HPGRL) algorithms for scaling PGRL methods to high-dimensional domains. In HPGRL, each subtask is defined as a PGRL problem whose solution involves computing a locally optimal policy. Subtasks are formulated in terms of a parameterized family of policies, a performance function, a method to estimate the gradient of the performance function, and a routine to update the policy parameters using this gradient. The usually slow convergence of HPGRL algorithms is improved by formulating high-level subtasks, which usually require low-resolution discretization of the state space and have finite action spaces, as value function RL problems, and lower-level subtasks, which usually require high-resolution discretization of the state space and may have infinite action spaces, as PGRL problems. This family of algorithms is termed **hierarchical hybrid** algorithms. The effectiveness of the proposed algorithms is demonstrated using a taxi-fuel problem as well as a more complex continuous state and action ship steering task.

This thesis also examines the use of HRL to accelerate policy learning in cooperative multi-agent tasks. The use of hierarchy speeds up learning in multi-agent domains by making it possible to learn coordination skills at the level of subtasks instead of primitive actions. Subtask-level coordination allows for increased cooperation skills as agents do not get confused by low-level details. A framework for hierarchical multi-agent RL is developed and an algorithm called **Cooperative HRL** is presented that solves cooperative multi-agent problems more efficiently. This algorithm is empirically evaluated using a

large four-agent AGV scheduling task. The framework and algorithm is extended to include communication decisions. The goal is for agents to learn both action and communication policies that together optimize the task given the communication cost. The extended algorithm, called **COM-Cooperative HRL**, is a hierarchical multi-agent RL algorithm with communication decisions. The efficacy of this algorithm as well as the relation between communication cost and the learned communication policy is demonstrated using a multi-agent taxi problem.

Together, the methods and algorithms developed in this dissertation use prior knowledge in a principled way, and extend HRL to solving complex sequential decision making problems such as those with continuous state and/or continuous action spaces and domains with multiple cooperative agents.

CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Sequential decision making under uncertainty is one of the fundamental problems in Artificial Intelligence (AI). Many sequential decision making problems can be modeled using the Markov decision process (MDP) formalism. An MDP (Howard, 1960; Puterman, 1994) models a system that we are interested in controlling as being in some state at each time step. As a result of actions the agent selects, the system moves through some sequence of states and receives a sequence of rewards. The goal is to select actions to maximize some measure of long-term reward.

Reinforcement learning (RL) is a machine learning framework for solving problems posed in the MDP formalism. Despite its numerous successes in a number of different domains, including backgammon (Tesauro, 1994), job-shop scheduling (Zhang and Dietterich, 1995), dynamic channel allocation (Singh and Bertsekas, 1996), elevator scheduling (Crites and Barto, 1998), and helicopter flight control (Ng et al., 2004), current RL methods do not scale well to high dimensional domains — they can be slow to converge and require too many training samples to be practical for many real-world problems. This issue is known as the **curse of dimensionality**: the exponential growth of the number of parameters to be learned with the size of any compact encoding of system state (Bellman, 1957). Recent attempts to combat the curse of dimensionality have turned to principled ways of exploiting abstraction in RL. This leads naturally to hierarchical control architectures and associated learning algorithms.

Although hierarchical reinforcement learning (HRL) approaches exploit the power of abstraction and scale better than flat RL methods to high dimensional domains, they still

suffer from the main limitation of flat RL algorithms: the *curse of dimensionality*. Moreover, HRL methods have so far only been studied in a narrow context: they have been investigated for the discrete-time discounted reward SMDP model; they have all been value function RL methods; and, they have only been studied in single-agent domains.

This dissertation expands the context and scope of HRL. The objective here is to develop several novel extensions to existing HRL frameworks and design algorithms that are appropriate for solving complex sequential decision making problems such as those with continuous state and/or continuous action spaces, and domains with multiple cooperative agents.

1.1 Motivation

Many problems faced by animals and AI systems can be modeled as sequential decision making in uncertain dynamic environments. For example, a complex manufacturing system, e.g., a system for manufacturing automobile or personal computers, involves optimizing hundreds or even thousands of processes (sub-systems) such as inventory, engineering design, assembly, and marketing.

These problems involve decision makers, or agents, selecting sequences of actions in order to achieve multiple long-term goals. Moreover, uncertainty is ever present in these domains, both in the effects of actions, and in the evolution of the actual system. The uncertain and ever changing nature of these problems makes it difficult to plan ahead of time. Hence, these tasks require control rules, which are dependent on the state variables of the system. In recent years, advances in technology have led to increased interest in automated methods for solving these tasks. Commercial tools are now available for problems ranging from factory optimization to medical diagnosis. Unfortunately, these problems tend to be very complex, and most of the existing automated techniques either build on heuristics, or do not fully address the long-term or the uncertain aspects of these sequential decision making tasks.

Fortunately, although such problems are very complex, they are often hierarchically decomposable into a set of simpler subtasks. As argued by Simon (1981) in "Architecture of Complexity," many complex systems have a decomposable hierarchical structure, with the subsystems interacting only weakly between themselves. Humans exploit this decomposable hierarchical structure in solving such complex and large-scale problems.

An example will help illustrate the basic concepts. This example has been chosen because, it involves an interesting and challenging manufacturing system, and furthermore several versions of this example have been used in the experiments of this dissertation. Figure 1.1 shows an automated guided vehicle (AGV) scheduling task. AGVs are used in flexible manufacturing systems (FMSs) for material handling (Askin and Standridge, 1993). They are typically used to pick up parts from one location and drop them off at another location for further processing. Locations correspond to workstations ($M1$ to $M4$) or storage locations (*load* and *unload* stations). Loads that are released at the drop-off points ($D1$ to $D4$) of workstations wait at their pick-up points ($P1$ to $P4$) after the processing is over, so the AGV is able to take them to the warehouse or some other locations. The pick-up points ($P1$ to $P4$) are the machine or workstations' output buffers. Any FMS using AGVs faces the problem of optimally scheduling the paths of the AGVs in the system (Klein and Kim, 1996). For example, a move request occurs when a part finishes at a workstation. If more than one vehicle is empty, the vehicle which would service this request needs to be selected. Also, when a vehicle becomes available, and multiple move requests are queued, a decision needs to be made as to which request should be serviced by that vehicle. These schedules obey a set of constraints that reflect the temporal relationships between activities and the capacity limitations of a set of shared resources. The system performance is generally measured in terms of throughput, on-line inventory, and AGV travel time, but throughput is by far the most important factor. Throughput is measured in terms of the number of finished assemblies deposited at the unloading deck per unit time. Since this problem is very complex, various heuristics and their combinations are generally used

to schedule AGVs (Klein and Kim, 1996). However, the heuristics perform poorly when the constraints on the movement of the AGVs are reduced.

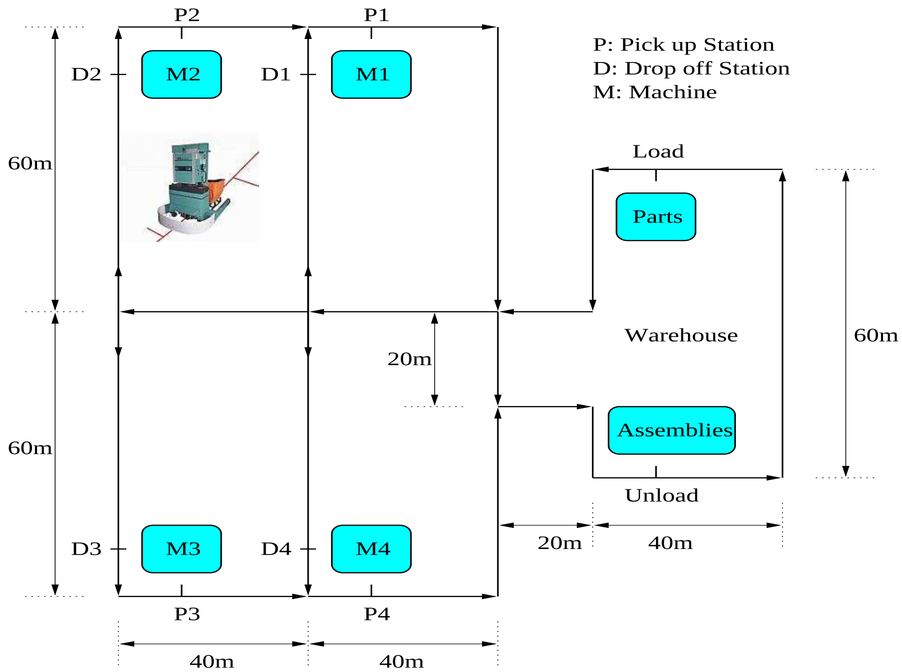


Figure 1.1. An AGV scheduling domain with four machines $M1$ to $M4$. AGVs are responsible to carry raw materials and finished parts between the machines and the warehouse.

In order for an AGV to optimize this task, it must learn all its sub-tasks such as carry parts from load station to machines, deliver assemblies from machines to unload station at the warehouse, navigate to load and unload stations, plus it should learn the order to execute these sub-tasks. The state space of this task consists of AGV's status and location, status of input and output buffers of workstations, and the availability of parts in warehouse, which can become enormous. It makes it very difficult for flat (non-hierarchical) RL methods to be used in this problem as we will show in Chapters 4 and 6.

However, the AGV scheduling task described above is naturally decomposed to a set of *non-primitive* subtasks like deliver material to workstations ($DM1$ to $DM4$), deliver assembly from workstations to warehouse ($DA1$ to $DA4$), navigate to the load station at the warehouse (*NavLoad*), navigate to the drop-off points of workstations (*NavPut1* to

NavPut4), navigate to the pick-up points of workstations (*NavPick1* to *NavPick4*), navigate to the unload station at the warehouse (*NavUnload*), and a set of *primitive* subtasks such as *load*, *put*, *pick*, *unload*, *left*, *forward*, and *right*. These are the subtasks that are naturally important in solving the AGV scheduling task. The designer of the system uses her/his domain knowledge to put the *primitive* and *non-primitive* subtasks of the AGV scheduling problem together and builds a hierarchical task decomposition like the one shown in Figure 1.2. This hierarchical decomposition can later be used by HRL algorithms such as hierarchy of abstract machines (HAMs) (Parr, 1998), options (Sutton et al., 1999; Precup, 2000), MAXQ (Dietterich, 2000), and programmable HAMs (PHAMs) (Andre and Russell, 2001; Andre, 2003) to optimize the AGV scheduling problem. Using of hierarchical RL methods leads to faster convergence and better performance than the flat algorithms as we will show for MAXQ in this thesis.

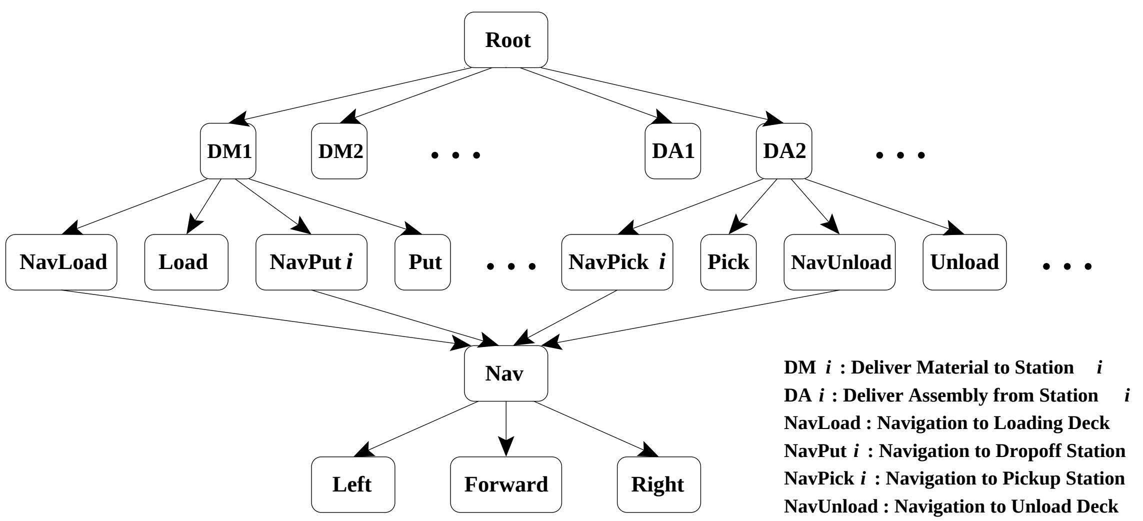


Figure 1.2. A hierarchical task decomposition for an AGV scheduling problem.

These HRL algorithms find the hierarchically or recursively optimal discounted reward policy for the AGV scheduling problem when the number of states is finite. However as we mentioned earlier, even HRL algorithms suffer from the *curse of dimensionality*. It will take a long time and require too many samples for them to converge if the state space of

the system grows. It raises several important questions such as: **1)** Is the discounted reward optimality the most suitable optimality criterion for this task? If it is not, is it possible to design HRL algorithms to find a more appropriate optimal policy for this problem? **2)** Consider the continuous state and action version of the AGV scheduling problem, when the AGV must learn to navigate using low-level continuous commands instead of directional actions such as *forward* or *left*, and it has continuous sensors instead of only viewing the world as a discrete grid. Are the existing HRL algorithms still able to solve the problem efficiently? **3)** Consider the multi-agent version of the AGV scheduling problem where there are several AGVs in the environment cooperating with each other to carry parts to workstations and bring assemblies from workstations back to the warehouse. The number of states and actions, and as a result the number of parameters to be learned, increases dramatically with the number of agents (AGVs). Does the nature of cooperative multi-agent problems allow us to design more efficient HRL algorithms for these domains? These are the types of the questions that we try to address in this dissertation. We briefly describe how we address the above questions in the next section, and leave the more elaborative discussion for later chapters.

1.2 Our Approach

Prior work in HRL including HAMs, options, MAXQ, and PHAMs has been limited to the discrete-time discounted reward SMDP model. However, the average reward optimality criterion is generally more appropriate in modeling cyclical control and optimization tasks, such as queuing, scheduling, and flexible manufacturing. We investigate two formulations of HRL based on the *average reward* SMDP model, both for discrete-time and continuous-time. These formulations correspond to two notions of optimality that have been previously explored in HRL: *hierarchical optimality* and *recursive optimality* (Dietterich, 2000). We present algorithms that learn to find hierarchically and recursively optimal average reward policies under discrete-time and continuous-time average reward SMDP models. We call

them **hierarchically optimal average reward RL** (HAR) and **recursively optimal average reward RL** (RAR) algorithms.

Existing HRL approaches are limited to value function RL (VFRL) methods. However, there are only weak theoretical guarantees on the performance of VFRL algorithms on problems with large or continuous state spaces. Policy gradient RL (PGRL) methods have demonstrated better performance in problems with continuous state and/or continuous action spaces (Marbach, 1998; Baxter et al., 2001). We propose a family of **hierarchical policy gradient RL** (HPGRL) algorithms that exploit both the power of abstraction, and the efficiency of PGRL methods in continuous state and/or continuous action problems. However, they suffer from slow convergence of PGRL algorithms. Consider the continuous state and action version of the AGV scheduling task again. The low-level subtasks such as *NavUnload* are now continuous state and action problems. The AGV needs to know its exact location and selects its action among infinite number of possibilities in order to solve these low-level continuous state and action subtasks. In contrast, when AGV decides at the high-level in the hierarchy, for instance to choose between delivering material to or from machines, it needs only a rough estimate of its location. Additionally, the AGV selects its action among only eight possible choices ($DM1$ to $DM4$ and $DA1$ to $DA4$). We accelerate learning of HPGRL algorithms by formulating high-level subtasks, which usually have smaller state and finite action spaces as VFRL problems, and low-level subtasks such as *NavUnload* with infinite state and/or action spaces as PGRL problems. We call this family of algorithms **hierarchical hybrid** algorithms.

Finally, we examine the use of HRL to accelerate policy learning in cooperative multi-agent tasks. The nature of cooperative multi-agent problems allows for more efficient use of HRL methods. Consider the multi-agent version of the AGV scheduling task again. In our approach, AGVs use the same hierarchical task decomposition. Learning is decentralized, with each agent learning three interrelated skills. First, how to perform subtasks such as deliver material to machine $M1$ ($DM1$) or navigation to unload station (*NavUnload*).

Second, the order to do the subtasks, for instance go to the load station and pick up part 1 before heading to workstation $M1$. Third, how to coordinate with other agents, AGV 1 can carry part for workstation $M1$ while AGV 2 makes the output buffer of $M1$ empty. The use of hierarchy allows AGVs to learn more efficiently by making it possible to learn coordination skills at the level of subtasks instead of primitive actions. Subtask-level coordination allows for increased cooperation skills as agents do not get confused by low-level details. Each AGV learns high-level coordination knowledge (e.g., what is the utility of AGV 1 carrying part to machine $M1$ if AGV 2 is bringing assembly back from machine $M3$), rather than it learns its response to low-level primitive actions of other AGVs (e.g., if AGV 1 goes forward, what should AGV 2 do).

In addition to the *curse of dimensionality*, multi-agent learning suffers from **partial observability**. Even if an agent has complete observability of its own state, states and actions of other agents are not fully observable. One way to address *partial observability* in distributed multi-agent domains is to use communication to exchange required information. However, communication is usually costly, which requires agents to optimize their communication policy in addition to their action policy. A further advantage of the use of temporal abstraction in cooperative multi-agent learning is that AGVs now communicate at the level of subtasks (temporally extended actions) instead of primitive actions. Since subtasks can take a long time to complete, communication is needed only fairly infrequently.

In this research, we introduce a hierarchical multi-agent RL framework and present two algorithms called **Cooperative HRL** and **COM-Cooperative HRL**. In **Cooperative HRL** algorithm, we assume communication is free. In **COM-Cooperative HRL** algorithm, we assume communication is costly, and agents learn both action and communication policies that together optimize the task given the communication cost. Of course, it makes **COM-Cooperative HRL** slower than **Cooperative HRL** due to more parameters that must be learned.

1.3 Contributions

The main contributions of this dissertation are summarized below.

Hierarchical Reinforcement Learning

- We have developed a general hierarchical reinforcement learning (HRL) framework for simultaneous learning of policies at multiple levels of the hierarchy. This framework is a generalization of existing HRL approaches especially the MAXQ value function decomposition (Dietterich, 2000). In our framework, we apply the three-part value function decomposition (Andre and Russell, 2002) to guarantee hierarchical optimality, and use reward shaping (Ng et al., 1999) to reduce the burden of exploration, thereby extending the MAXQ method.

Hierarchical Average Reward Reinforcement Learning

- We extend previous work on hierarchical reinforcement learning (HRL) to the average reward SMDP model, and investigate hierarchical and recursive optimalities in hierarchical average reward RL.
  - We have developed new discrete-time and continuous-time *hierarchically optimal average reward RL* (HAR) algorithms. The aim of these algorithms is to find a hierarchical policy with highest *global gain*.
  - We have developed new discrete-time and continuous-time *recursively optimal average reward RL* (RAR) algorithms. In these algorithms, we treat subtasks as continuing average reward problems, where the goal at each subtask is to maximize its gain given the policies of its children. We investigate the optimality achieved by the RAR algorithm and illustrate the conditions under which the policy learned by this algorithm at each subtask is independent of the context in which it is executed and therefore can be reused by other hierarchies.

- We empirically demonstrate the effectiveness and the type of optimality achieved by HAR and RAR algorithms using two AGV scheduling tasks.

Hierarchical Policy Gradient Reinforcement Learning

- We have developed a family of *hierarchical policy gradient RL* (HPGRL) algorithms for scaling policy gradient reinforcement learning methods to problems with continuous (or large discrete) state and/or action spaces.

- We present a family of *hierarchical hybrid* algorithms to accelerate learning in HPGRL algorithms. In *hierarchical hybrid* algorithms, we formulate high-level subtasks, which usually require low-resolution discretization of the state space and have finite action spaces as value function RL problems, and low-level subtasks, which usually require high-resolution discretization of the state space and may have infinite action spaces as policy gradient RL problems.

- We empirically demonstrate the performance of *hierarchical hybrid* algorithms using a continuous state and action ship steering problem.

Hierarchical Multi-Agent Reinforcement Learning

- We extend the SMDP model to cooperative multi-agent domains and present the *multi-agent SMDP* (MSMDP) model.

- We have developed a hierarchical cooperative multi-agent RL framework in which agents learn coordination faster by sharing information at the level of subtasks, rather than attempting to learn coordination at the level of primitive actions.

- We employ this hierarchical cooperative multi-agent RL framework, and present a hierarchical multi-agent RL algorithm called *Cooperative HRL*.

- We empirically demonstrate the effectiveness of the *Cooperative HRL* algorithm using a large four-agent AGV scheduling problem.

- We extend the *Cooperative HRL* algorithm to include communication decisions, and present a hierarchical multi-agent RL algorithm called *COM-Cooperative HRL*. This algorithm is designed to learn both action and communication policies that together optimize the task given the communication cost.

- We empirically demonstrate the effectiveness of the *COM-Cooperative HRL* algorithm using a multi-agent taxi problem.

1.4 Outline

The remainder of this thesis is organized as follows:

Chapter 2: We present the foundational background material for the dissertation. We begin by describing the reinforcement learning (RL) problem and formalizing the Markov decision process (MDP) and semi-Markov decision process (SMDP) frameworks under different optimality criteria. We also review some of the key ideas and solution methods of MDPs and SMDPs. We discuss some of the difficulties of solving MDPs for problems with large state spaces. Then we briefly review the historical development of hierarchy and temporal abstraction in artificial intelligence (AI), control theory, and RL. In this, we especially emphasize hierarchical reinforcement learning (HRL) and the main concepts and algorithms in this framework. Finally, we present a brief overview of the growing field of multi-agent reinforcement learning. In this chapter, we also introduce the notation that will be used in this dissertation.

Chapter 3: We present a general framework for hierarchical reinforcement learning (HRL) which is used in the algorithms proposed in this dissertation. We also illustrate the basic concepts of HRL such as policy execution, hierarchical and recursive optimality, and value function definitions and decompositions in this chapter.

Chapter 4: We present *hierarchically optimal average reward RL* (HAR) and *recursively optimal average reward RL* (RAR) algorithms for both discrete and continuous time SMDP models. We investigate the conditions under which the policy learned by the RAR algorithm at each subtask is independent of the context in which it is executed and therefore can be reused by other hierarchies. We use two AGV tasks to demonstrate the performance and the type of optimality achieved by these algorithms.

Chapter 5: We first present a family of *hierarchical policy gradient RL* (HPGRL) algorithms and compare their performance with hierarchical value function RL (VFRL) algorithms in a simple taxi-fuel problem. We then show how learning can be accelerated in HPGRL algorithms by using both value function and policy gradient RL formulations in a hierarchy, and propose a family of *hierarchical hybrid* algorithms. We empirically demonstrate the performance of a *hierarchical hybrid* algorithm using a continuous state and action ship steering problem.

Chapter 6: We investigate the use of hierarchical reinforcement learning (HRL) to speed up the acquisition of cooperative multi-agent tasks. We first extend the SMDP model to cooperative multi-agent domains and present the *multi-agent SMDP* (MSMDP) model. We use this model and present a hierarchical cooperative multi-agent RL framework. We then use this hierarchical cooperative multi-agent RL framework, and propose two hierarchical cooperative multi-agent RL algorithms called *Cooperative HRL* and *COM-Cooperative HRL*. While the *Cooperative HRL* algorithm assumes that communication is free, in the *COM-Cooperative HRL* algorithm, agents learn both action and communication policies that together optimize the task given the communication cost. The effectiveness of the *Cooperative HRL* algorithm is empirically demonstrated using a large four-agent AGV scheduling problem. We also empirically demonstrate the efficacy of the *COM-Cooperative HRL* algorithm as well as the relation between the communication cost and the learned

communication policy using a multi-agent taxi problem.

Chapter 7: We summarize the dissertation and discuss directions for future research.

Appendix: We define a table of the symbols used in this dissertation.

CHAPTER 2

BACKGROUND AND NOTATION

In this chapter, we describe the reinforcement learning (RL) problem and introduce the Markov decision process (MDP) and semi-Markov decision process (SMDP) formalisms under different optimality criteria. We also present some of the key ideas and solution methods of MDPs and SMDPs. Then we review the historical development of hierarchy and temporal abstraction in artificial intelligence (AI), control theory, and RL. In this, we especially emphasize hierarchical reinforcement learning (HRL) and the main concepts and algorithms in this framework. Finally, we present a brief overview of the growing field of multi-agent reinforcement learning. In doing so, we also introduce the notation that will be used in the remainder of this dissertation.

Throughout this chapter we present the standard body of background work in the field. For more comprehensive introduction to MDPs, SMDPs, and RL, readers may also refer to standard texts such as (Howard, 1960, 1971; Puterman, 1994; Bertsekas, 1995; Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 1998) or the survey by Kaelbling et al. (Kaelbling et al., 1996). Barto and Mahadevan (2003) provides more detailed introduction to HRL.

2.1 Reinforcement Learning

Reinforcement learning (RL) (Sutton and Barto, 1998) refers to a collection of methods that allow an agent (a system) to learn how to make good decisions by observing its own behavior, and improves its actions through a reinforcement mechanism. There are many formal specifications of this kind of problems that have been developed over the last fifty years. The most commonly used is the **Markov decision processes** (MDPs). An

MDP assumes that the agent has full access to the state of the world and each of its actions takes a single time step. **Semi-Markov decision processes** (SMDPs) relax the latter assumption and allow actions that take several time steps. Finally, **partially observable Markov decision processes** (POMDPs) relax the former assumption by allowing the agent to receive observations that do not necessarily reveal the entire state of the environment. When a problem is modeled using one of the above, the goal of an RL method is to find a good (possibly optimal) policy for the model. We will cover MDPs and SMDPs in detail in Sections 2.2 and 2.3. POMDPs will be presented more briefly, as the subject of partial observability is almost (but not completely) orthogonal to the main contributions of this dissertation.

2.2 Markov Decision Processes

Markov decision processes (MDPs) (Howard, 1960; Puterman, 1994) are model for sequential decision making when outcomes are uncertain. There are many possible ways of defining MDPs, and many of these definitions are equivalent up to small transformations of the problem. One definition is that an MDP model $\mathcal{M}$ consists of five elements $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, I \rangle$ defined as follows:[1]

- $\mathcal{S}$: is the set of **states** of the world.
- $\mathcal{A}$: is the set of possible **actions** from which the agent (controller) may choose on at each decision epoch.
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$: is the **transition probability function** with $P(s'|s, a)$ being the probability of transition to state s' when agent takes action a in state s.

[1] In non-discrete settings (when the set of states $\mathcal{S}$ and the set of actions $\mathcal{A}$ are not discrete), many subtle mathematical issues arise, which are not in the scope of this dissertation. For more details see (Howard, 1960; Puterman, 1994).

- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$: is the **reward function** with $r(s, a)$ being the reward that agent receives when it takes action a in state s.

- $I : \mathcal{S} \rightarrow [0, 1]$: is the **initial state distribution**.

The qualifier "Markov" is used because the transition probability and reward functions depend on the past only through the current state of the system and the action selected by the decision maker in that state. Since it may not be possible for the agent to take every action at each state s, we define $\mathcal{A}_s \subseteq \mathcal{A}$ as the set of admissible actions in state s. Events in an MDP proceed as follows. The agent begins in an initial state s_0 drawn from the initial distribution I. At each time t, the agent observes the state of the environment $s_t \in \mathcal{S}$, selects an action $a_t \in \mathcal{A}_{s_t}$, as a result of which the state of the system transitions to some state $s_{t+1} \in \mathcal{S}$ drawn from the transition probability function $P(s_{t+1}|s_t, a_t)$, and the agent receives reward $r(s_t, a_t)$.

The method of specifying an agent's behavior in an MDP is called a **policy**. A policy can be **stationary**, in which case it is a stochastic mapping from states to actions, but it can also be **non-stationary** and depend on other factors such as the agent's memory or internal state. A stationary policy, μ, can be **deterministic**, in which case it is a mapping from states to actions $\mu : \mathcal{S} \rightarrow \mathcal{A}$, or **stochastic**, in which case it is a probability distribution over state-action pairs $\mu : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$ and $\sum_{a \in \mathcal{A}_s} \mu(a|s) = 1$ for all $s \in \mathcal{S}$, where $\mu(a|s)$ represents the probability that policy μ selects action a in state s.

Now the question arises of the quality of a given policy. There are many ways of defining optimality, but typically the quality or value of a policy is based on a function of the future rewards. In Sections 2.2.1, 2.2.2, and 2.2.3, we examine several popular optimality criteria in the MDP literature.

2.2.1 Undiscounted Reward Markov Decision Processes

In **episodic tasks**, the environment has one or more **absorbing** terminal states. All transitions from an absorbing terminal state lead back into the same state with probability 1.0

and reward 0. Typically in this setting, the goal is to maximize the expected **undiscounted** sum of rewards $\sum_{t=0}^{N-1} r(s_t, a_t)$, where N is the number of time steps taken before reaching an absorbing state. We usually consider only policies that are **proper** in that all policies reach an absorbing terminal state with probability 1.0 (Bertsekas and Tsitsiklis, 1996).

In **infinite-horizon** setting where the agent may take an infinite number of steps, the undiscounted sum of rewards can be infinite. To avoid this, discounted and average reward optimality criteria are often used, which we describe them in the next two sections.

2.2.2 Discounted Reward Markov Decision Processes

In **discounted reward** MDPs, near-term rewards are weighted more than distant rewards. In this setting, the agent's goal is to maximize $\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)$. This sum is finite if the **discount factor** $0 \leq \gamma < 1$, and all rewards are bounded. Note that the episodic problems can be folded into this setting — if all policies are proper and we use a discount factor of $\gamma = 1$, the undiscounted sum of rewards of an episodic task remains finite (Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 1998).

In the infinite-horizon discounted reward setting, the **value function** for a policy μ, $V^\mu : \mathcal{S} \rightarrow \mathbb{R}$, is a mapping from states to their values under policy μ. The value of state s under policy μ expresses the expected discounted sum of future rewards starting from state s and following policy μ thereafter. Formally, we define the value function of a policy as

$$
\begin{aligned}
V^\mu(s) =& E\left[r(s_0, \mu(s_0)) + \gamma r(s_1, \mu(s_1)) + \gamma^2 r(s_2, \mu(s_2)) + \ldots | s_0 = s, \mu\right] \\
=& E\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) | s_0 = s, \mu\right]
\end{aligned}
$$

We can relate the values of different states using what are known as the **Bellman equations** (Bellman, 1957). These equations relate each state to its possible successor states.

$$
V^\mu(s) = \sum_{a \in \mathcal{A}_s} \mu(a|s) \left[r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s, a) V^\mu(s')\right] \tag{2.1}
$$

A policy μ is optimal if, for all states, its value is at least as high as the value of any other policy. It is known (Blackwell, 1962) that there exists a deterministic optimal policy for infinite-horizon discounted reward MDPs. The **optimal policy** μ^* is specified as

$$\mu^*(s) = \arg\max_{a \in \mathcal{A}_s} \left[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^*(s') \right]$$

where V^* is the **optimal value function**, the value function of the optimal policy. Bellman proved that the optimal value function is the solution to the following equation:

$$V^*(s) = \max_{\mu} V^{\mu}(s) = \max_{a \in \mathcal{A}_s} \left[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^*(s') \right] \tag{2.2}$$

Similarly, the **action-value function** of a policy μ, $Q^{\mu} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, is defined as a mapping from state-action pairs to their values. The action-value function $Q^{\mu}(s,a)$ for a policy μ is the expected sum of discounted future rewards for taking action a in state s and then following policy μ.

$$Q^{\mu}(s,a) = E\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) | s_0 = s, a_0 = a, \mu \right]$$

Note that $V^{\mu}(s) = Q^{\mu}(s, \mu(s))$. The Bellman equation for the action-value function Q^{μ} can be written as

$$Q^{\mu}(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \sum_{a' \in \mathcal{A}_{s'}} \mu(a'|s') Q^{\mu}(s', a')$$

and the **optimal action-value function** Q^* is the solution to the Bellman optimality equation for action-value function defined as follows:

$$Q^*(s,a) = \max_{\mu} Q^{\mu}(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \max_{a' \in \mathcal{A}_{s'}} Q^*(s', a') \tag{2.3}$$

The Bellman Equations 2.2 and 2.3 are related by:

$$V^*(s) = \max_{a \in \mathcal{A}_s} Q^*(s, a)$$

An alternative way of defining the optimal value function is based on the **Bellman operator** Γ^* (Bertsekas, 1995) defined as

$$\Gamma^* V^\mu(s) = \max_{a \in \mathcal{A}_s} Q^\mu(s, a)$$

The optimal value function V^* is the fixed point of $V^* = \Gamma^* V^*$.

2.2.3 Average Reward Markov Decision Processes

Discounted optimization is motivated by domains where reward can be interpreted as money that can earn interest, or where there is a fixed probability that a run will be terminated at any given time. However, many domains do not have either of these properties. Discounting in such domains tends to sacrifice long-term rewards in favor of short-term rewards. Moreover, in general, the discounted optimal policy depends on the choice of the value of the discount factor γ. For instance, consider the MDP of Figure 2.1 from Schwartz (1993). Here, any undiscounted reward method will clearly choose action a_1 in state s_1. But for any $\gamma < \frac{500}{501} \approx 0.998$, $Q^\mu(s_1, a_2) > Q^\mu(s_1, a_1)$ regardless of policy μ. In fact, given any γ, there is some value we can set for $r(s_1, a_2)$ which makes the discounted criterion favor action a_2 over action a_1.

It is true that for any finite MDP (an MDP with finite state and action spaces) there is some sufficiently large γ for which the discounted and undiscounted measures agree. However, proper choice of such a γ requires detailed knowledge of the domain — the knowledge that we do not want to presuppose. Even, with such knowledge, a parameter such as γ that needs to be tailored to suit individual domains is clearly undesirable. Therefore, the agent may prefer to compare policies on the basis of their average expected reward instead of

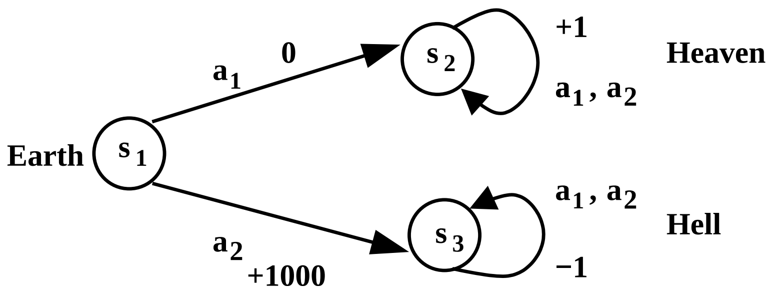


Figure 2.1. An MDP on which discounted and undiscounted measures may disagree.

their expected discounted reward. The aim of the average reward MDP is to compute policies that yield the highest expected payoff per step. The **average reward** or **gain** associated with a particular policy μ, g^μ, is defined as

$$g^\mu(s) = \lim_{N\to\infty} \frac{1}{N} E\left[\sum_{t=0}^{N-1} r(s_t, \mu(s_t))|s_0 = s, \mu\right] \tag{2.4}$$

when the state space of the MDP, $\mathcal{S}$, is finite or countable, Equation 2.4 can be written as

$$g^\mu(s) = \lim_{N\to\infty} \frac{1}{N} \sum_{t=0}^{N-1} (\boldsymbol{P}^\mu)^t r(s, \mu(s)) = \bar{\boldsymbol{P}}^\mu r(s, \mu(s)) \tag{2.5}$$

where $\boldsymbol{P}^\mu$ and $\bar{\boldsymbol{P}}^\mu = \lim_{N\to\infty} \frac{1}{N} \sum_{t=0}^{N-1} (\boldsymbol{P}^\mu)^t$ are the transition probability matrix and the *limiting matrix* of policy μ respectively.[2]

A key observation that greatly simplifies the design of the average reward algorithms is that for **unichain** MDPs,[3] the average reward of any policy is state independent, that is $g^\mu(s) = g^\mu, \forall s \in \mathcal{S}$. From now on in this section we assume that MDPs are *unichain*.

---

[2]The *limiting matrix* $\bar{\boldsymbol{P}}$ satisfies the equality $\boldsymbol{P}\bar{\boldsymbol{P}} = \bar{\boldsymbol{P}}$.

[3]MDPs in which every stationary policy gives rise to a Markov chain with a single recurrent class.

In average reward MDP, a policy μ is measured using a different value function, namely the average-adjusted sum of rewards earned following that policy.[4]

$$H^{\mu}(s) = \lim_{N\to\infty} E\left\{\sum_{t=0}^{N-1}\left[r(s_t,\mu(s_t)) - g^{\mu}\right]|s_0 = s, \mu\right\}$$

The term H^{μ} is usually referred to as the **average-adjusted value function**. Furthermore, the average-adjusted value function satisfies the Bellman equation

$$H^{\mu}(s) + g^{\mu} = r(s,\mu(s)) + \sum_{s'\in\mathcal{S}} P(s'|s,\mu(s))H^{\mu}(s')$$

Similarly, the **average-adjusted action-value function** for a policy μ, L^{μ}, is defined, and it satisfies the Bellman equation

$$L^{\mu}(s,a) + g^{\mu} = r(s,a) + \sum_{s'\in\mathcal{S}} P(s'|s,a)L^{\mu}(s',\mu(s'))$$

We define a **gain-optimal** policy μ^* as one that has the maximum average reward over all policies, that is $g^* \geq g^{\mu}$. The gain-optimal policy satisfies the following Bellman optimality equations for average-adjusted value function and average-adjusted action-value function (Bertsekas, 1995).

$$H^*(s) + g^* = \max_{a\in\mathcal{A}_s}\left[r(s,a) + \sum_{s'\in\mathcal{S}} P(s'|s,a)H^*(s')\right] \tag{2.6}$$

$$L^*(s,a) + g^* = r(s,a) + \sum_{s'\in\mathcal{S}} P(s'|s,a)\max_{a'\in\mathcal{A}_{s'}} L^*(s',a') \tag{2.7}$$

It is proved (Howard, 1960; Puterman, 1994) that for any unichain MDP, there exist a g^* and a function H^* over $\mathcal{S}$ that satisfy the Equation 2.6 (or a function L^* over $\mathcal{S}\times\mathcal{A}$ that satisfies

[4] This limit assumes that all policies are aperiodic. For periodic policies, it changes to the Cesaro limit $H^{\mu}(s) = \lim_{N\to\infty}\frac{1}{N}\sum_{k=0}^{N-1} E\left\{\sum_{t=0}^{k}\left[r(s_t,\mu(s_t)) - g^{\mu}\right]|s_0 = s,\mu\right\}$ (Puterman, 1994).

the Equation 2.7). Further, g^*, H^*, and L^* are gain, average-adjusted value function, and average-adjusted action-value function of the gain-optimal policy μ^*.

2.2.4 Solution Methods for MDPs

Now that we have defined the MDP model, the next task is to solve it, i.e., to find an optimal policy and/or the optimal value function.[5] There are variety of methods for achieving this. Some methods require knowing the transition probability and reward functions and are performed without access to an environment; these are considered **offline** algorithms. These are the standard **dynamic programming** (DP) algorithms from the field of operations research. Having the model allows the simulation of the domain so as to do **planning** to find the optimal value function and/or an optimal policy without interacting directly with the environment. Other methods work without assuming prior knowledge of the model and operate by learning through experience in the environment; these are called **online** algorithms.

Since a value function (or an action-value function) defines a policy in an MDP, one approach to find the optimal policy is to compute the optimal value (action-value) function first, and then extract the optimal policy from it. We call the algorithms utilizing this approach, value function algorithms. Another approach is to directly find the optimal policy. The methods using this approach are called policy search methods. In Sections 2.2.4.1 and 2.2.4.2, we present a brief overview of the above two approaches to solve an MDP model.

2.2.4.1 Value Function Solution Methods for MDPs

Value function (VF) methods attempt to find the optimal value (action-value) function and then extract an optimal policy from it. These algorithms have been extensively studied in the machine learning literature (Bertsekas and Tsitsiklis, 1996; Sutton and Barto,

[5]What we really mean by an optimal policy in this section is a reasonably good policy. Since in any real-world AI problem it is not possible to even imagine fi nding optimal policies.

1998) and have yielded some remarkable empirical successes in a number of different domains, including learning to play checkers (Samuel, 1959), backgammon (Tesauro, 1994), job-shop scheduling (Zhang and Dietterich, 1995), dynamic channel allocation (Singh and Bertsekas, 1996), and elevator scheduling (Crites and Barto, 1998). We now briefly review some standard VF algorithms.

If the model is known, then Equation 2.2 defines a system of equations, the solution to which yields the optimal value function. These equations may either be solved directly via solving a related linear program (e.g., Gordon (1999); de Farias (2002)), or by iteratively performing the update

$$V(s) = \max_{a \in \mathcal{A}_s} \left[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V(s') \right]$$

until it converges. The latter of these is called **value iteration** (Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 1998), which is a DP-based algorithm.

Another standard DP-based algorithm is **policy iteration** (Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 1998). It uses a policy μ and its estimated value function V, and iteratively updates μ according to

$$\mu(s) = \arg\max_{a \in \mathcal{A}_s} \left[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V(s') \right]$$

and updates V to be the value function V^μ for policy μ by solving the system of linear equations given by Equation 2.1.

Other instances of offline VF algorithms are **asynchronous value iteration** and **asynchronous policy iteration** (Bertsekas and Tsitsiklis, 1996; Sutton and Barto, 1998).

If the agent does not know the model of the domain, we may first try to interact with the environment to learn a model which is then used to compute optimal policies (e.g., Dyna (Sutton, 1991) and prioritized sweeping (Moore and Atkeson, 1993)). This is known as **Model-based** approach. Alternatively, we may try to learn the value (action-value) function

directly and do not explicitly learn a model. This approach is referred to as **model-free**, in that the agent does not need to learn the transition probabilities. Most of the model-free VF algorithms are instances of the **temporal difference** (TD) learning (Sutton, 1988), where the agent updates estimates of the value (action-value) function based in part on other estimates, without waiting for the true value. Two more popular TD methods are SARSA (Rummery and Niranjan, 1994) and Q-learning (Watkins, 1989).

The SARSA algorithm performs the following update upon seeing a transition from state s to s' when taking action a:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha\left[r(s,a) + \gamma Q(s',a')\right]$$

where α is called the learning rate parameter. SARSA causes action-value function Q to converge to the optimal action-value function, if a GLIE (Greedy in the Limit with Infinite Exploration) policy is used (Singh et al., 2000a). SARSA is known as an **on-policy** method, in that learns about the policy that it executes.

The Q-learning algorithm performs the following update when the agent takes action a in state s and transitions to state s':

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha\left[r(s,a) + \gamma \max_{a'\in\mathcal{A}_{s'}} Q(s',a')\right]$$

It can be shown that Q-learning converges with probability 1.0, if the agent uses an exploration policy that takes every state infinitely often and α satisfies some conditions (Jaakkola et al., 1994; Bertsekas and Tsitsiklis, 1996). Q-learning is known as an **off-policy** algorithm, meaning that the agent does not have to follow the policy for which it is learning a value function. This is advantageous in that a wider set of exploration methods are allowed.

Although most of the VF algorithms have been focused on the discounted setting, average reward VF methods have also been well studied. An average reward VF method is

an undiscounted infinite-horizon method for finding gain-optimal policies of an MDP (Mahadevan, 1996). It is generally appropriate in modeling cyclical control and optimization tasks, such as queuing, scheduling, and flexible manufacturing (Gershwin, 1994; Puterman, 1994). Several different types of average reward VF algorithms have been developed including offline algorithms such as (Bertsekas, 1998), model-based online methods such as (Tadepalli and Ok, 1998), discrete-time model-free online algorithms (Schwartz, 1993; Mahadevan, 1996; Tadepalli and Ok, 1996; Abounadi et al., 2001), and continuous-time model-free online algorithms (Mahadevan et al., 1997b; Wang and Mahadevan, 1999).

The discussion so far assumes that the state space $\mathcal{S}$ is sufficiently small that V can be stored explicitly as a table, with one entry for each state. For larger MDPs, these methods can be intractable. Specifically, in many problems, the number of states grows exponentially in the number of state variables. Similarly, if we apply grid-based discretization to an n-dimensional continuous state space to reduce the problem, we again end up with a number of discretized states that is exponential in n. Bellman called this problem the **curse of dimensionality** (Bellman, 1957), and it makes the straightforward application of RL algorithms impractical even for many moderate-dimensional problems.

Thus, in domains with large or infinite state spaces, one looks for approximation techniques that are based on a parametric representation of value function, rather than exact representation. A few examples of previous work proposing various approaches for doing so in different settings include (Van-Roy, 1998; Gordon, 1999; Koller and Parr, 2000; Guestrin et al., 2001; Dietterich and Wang, 2002; de Farias, 2002), and this topic remains an area of active research. The approximation methods have had some prominent empirical successes as mentioned at the beginning of this section. Despite numerous successes, the application of VF methods becomes problematic in domains with large or infinite state spaces. This is mainly because most algorithms for parametrically approximating value functions suffer from the following theoretical flaw: the performance of the policy derived from the approximate value function is not guaranteed to improve on each iteration, and in

fact can be worse than the policy in the previous iteration. This can happen even when the chosen parametric class contains a value function whose derived policy is optimal (Baxter and Bartlett, 2001). Additionally, VF methods become problematic when the state is only partially observable, because most methods for value function estimation critically rely on the Markov property. In the next section, we will describe an alternative approach to VF which addresses some of the above issues, and the problems that may happen when they are employed in complex domains.

2.2.4.2 Policy Search Solution Methods for MDPs

An alternative approach that circumvents the problems of VF methods mentioned at the end of Section 2.2.4.1 is to directly search in the space of policies. The methods using this approach to solve an MDP are known as **policy search** (PS) methods.[6]

PS methods have received much recent attention as a mean to solve problems with large or infinite state spaces, and problems with partially observable states. The motivation for this is three fold. **1)** For many MDPs, the value and action-value functions can be difficult to approximate, even though there may be simple and compactly representable policies that perform very well. Indeed, the existence of a good, compact representation of an action-value function implies the existence of a good, compact representation of a policy, because an action-value function defines a policy. In contrast, there is no guarantee that the existence of a good, compact representation of a policy implies a good, compact representation of an action-value function. **2)** Because PS algorithms start with a parameterized policy, it is relatively simple to choose a policy which incorporates prior knowledge via an appropriate choice of the parametric form of the policy. The use of prior knowledge in VF algorithms is not as easily realized. Finally, **3)** many real domains are only partially observable, and VF algorithms are known to be difficult to implement in such domains. Conversely, PS

[6] Policy iteration can also be considered a policy search (PS) method. However, since it uses value function, we categorize it as a value function (VF) method.

algorithms have been shown to work more effectively in partially observable domains. We might use a class of policies that depend only on the observables. This results in a class of memoryless (reactive) policies that can be applied to POMDP models (Williams and Singh, 1999). We can also introduce memory variables into the process state, and define limited memory policies (Mealeau et al., 1999). It permits belief state tracking, in which the agent uses past and present observations to estimate the true state.

Of course, while PS methods provide a powerful tool for solving many problems in RL and control, there are also settings in which VF algorithms may be preferred. For instance, explicitly searching in a policy space for a good policy may be computationally expensive and more prone to local optima than certain VF methods. So, if there is reason to believe that the value function can be easily approximated, then the VF approach would perhaps be method of choice. Moreover, if we do not have a prior knowledge about a likely form of a good policy, then one may instead use a VF algorithm.

A well-known class of PS algorithms are **policy gradient** (PG) algorithms. In these methods, we usually consider a class of parameterized stochastic policies, estimate the gradient of a performance function (e.g., average reward over time or weighted reward-to-go) with respect to policy parameters, and then improve the policy by adjusting the parameters in the direction of the gradient (Williams, 1992; Kimura et al., 1995; Marbach, 1998; Baxter et al., 2001). This approach has a long history in operations research, statistics, and control, forming the basis of perturbation analysis of discrete event dynamic systems (Ho and Cao, 1991; Cassandras and Lafortune, 1999). In addition to the pros and cons of PS methods mentioned above, one advantage of PG algorithms compared to VF methods is that they are theoretically guaranteed to converge to locally optimal policies, whereas VF algorithms can find globally optimal solutions. However, in practice it is usually not feasible to converge to globally optimal solutions in large domains in any case. However, PG methods usually suffer from the following problems. **1)** They may require up to an amount of sampling/number of steps that is exponential in the number of states or in the

horizon time. **2)** They are also limited to stochastic policies. In some domains, it seems very undesirable to add extra randomness to an already stochastic problem by forcing our policy to randomly choose its actions. **3)** They generally sample from the MDP once to take a small uphill step and then throw away the data.

One way to address some of the issues of using PG methods is to assume that the learning algorithm has access to the MDP via a generative model or a simulator (Kearns et al., 2000; Ng and Jordan, 2000; Ng, 2003). Ng et al. (2004) recently showed a very impressive application of this type of PS methods to autonomous helicopter flight.

2.3 Semi-Markov Decision Processes

Semi-Markov decision processes (SMDPs) (Howard, 1971; Puterman, 1994) extend the MDP model by allowing actions that take multiple time steps to complete. The action duration can depend on the transition that is made.[7] The state of the system may change continually between actions, unlike MDPs where state changes are only due to actions. Thus, SMDPs have become the preferred language for modeling temporally extended actions (Mahadevan et al., 1997a), which makes them very appealing in the context of hierarchical reinforcement learning, as we will see in Section 2.4.3.

An SMDP is defined as a five tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, I \rangle$. All components are defined as in an MDP except the transition probability function and the reward function. The transition probability function $\mathcal{P}$ now takes the duration of the actions into account. The transition probability function $\mathcal{P} : \mathcal{S} \times \mathbb{N} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is a multi-step transition probability function, with $P(s', N|s, a)$ denotes the probability that action a will cause the system to transition from state s to state s' in N time steps. This transition is at decision epochs only. Basically, the SMDP model represents snapshots of the system at decision points, whereas the so-called **natural process** describes the evolution of the system over all times. If we

[7]We are thus dealing with discrete-time SMDPs. Continuous-time SMDPs typically allow arbitrary continuous action durations.

marginalize $P(s', N|s,a)$ over N, we will obtain $m(s'|s,a)$ the transition probability for the embedded MDP. The term $m(s'|s,a)$ denotes the probability that the SMDP occupies state s' at the next decision epoch, given that the decision maker chooses action a in state s at the current decision epoch. The key difference in the reward function for SMDPs is that the rewards can accumulate over the entire duration of an action. As a result, SMDP reward for taking an action in a state depends on the evolution of the system during the execution of the action. Formally, SMDP reward is modeled as a function from $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$, with $r(s,a)$ represents the expected total reward between two decision epochs, given that the system occupies state s at the first decision epoch and the agent chooses action a. This expected reward contains all necessary information about the reward to analyze the SMDP model.

For each transition in an SMDP, the expected number of time steps until the next decision epoch is defined as

$$y(s,a) = E[N|s,a] = \sum_{N \in \mathbb{N}} N \sum_{s' \in \mathcal{S}} P(s', N|s,a)$$

The notions of policies and the various forms of optimality are the same for SMDPs as for MDPs. In infinite-horizon SMDPs, our goal is still to find a policy that maximizes either the expected discounted reward or the average expected reward. These two optimality criteria for an SMDP model will be discussed in sections 2.3.1 and 2.3.2.

2.3.1 Discounted Reward Semi-Markov Decision Processes

Recall that for a discounted MDP model, we expressed the expected value for following a policy as $E\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, \mu(s_t))|\mu\right]$. In discounted SMDP, because actions can take variable amounts of time, the value of a state s under a policy μ is defined as follows:

$$V^{\mu}(s) = E\left[r(s_0, \mu(s_0)) + \gamma^{N_0} r(s_1, \mu(s_1)) + \gamma^{N_0+N_1} r(s_2, \mu(s_2)) + \ldots |s_0 = s, \mu\right]$$

Now we can express the Bellman equations for discounted SMDPs as

$$V^{\mu}(s) = r(s, \mu(s)) + \sum_{s' \in \mathcal{S}, N \in \mathbb{N}} \gamma^N P(s', N | s, \mu(s)) V^{\mu}(s')$$

$$Q^{\mu}(s, a) = r(s, a) + \sum_{s' \in \mathcal{S}, N \in \mathbb{N}} \gamma^N P(s', N | s, a) Q^{\mu}(s', \mu(s'))$$

Similarly, we can write the Bellman optimality equations defining the optimal value function and optimal action-value function as

$$V^*(s) = \max_{a \in \mathcal{A}_s} \left[r(s, a) + \sum_{s' \in \mathcal{S}, N \in \mathbb{N}} \gamma^N P(s', N | s, a) V^*(s') \right]$$

$$Q^*(s, a) = r(s, a) + \sum_{s' \in \mathcal{S}, N \in \mathbb{N}} \gamma^N P(s', N | s, a) \max_{a' \in \mathcal{A}_{s'}} Q^*(s', a')$$

2.3.2 Average Reward Semi-Markov Decision Processes

The theory of infinite-horizon SMDPs with the average reward criterion is more complex than that for discounted models (Howard, 1971; Puterman, 1994). To simplify exposition we consider only unichain SMDPs. Under this assumption, the gain of any policy is state independent similar to the average reward MDP model.

The average expected reward or gain for a policy μ, g^{μ}, can be defined by taking the ratio of the expected total reward and the expected total number of time steps.

$$g^{\mu} = \liminf_{n \to \infty} \frac{E\left[\sum_{t=0}^{n-1} r(s_t, \mu(s_t)) | \mu\right]}{E\left[\sum_{t=0}^{n-1} N_t | \mu\right]} \tag{2.8}$$

where N_t is the total number of time steps until the next decision epoch, when agent takes action $\mu(s_t)$ in state s_t. When the state space of the SMDP, $\mathcal{S}$, is finite or countable, Equation 2.8 can be written as[8]

$$g^{\mu} = \frac{\bar{\boldsymbol{m}}^{\mu} r(s,\mu(s))}{\bar{\boldsymbol{m}}^{\mu} y(s,\mu(s))} \tag{2.9}$$

where $\boldsymbol{m}^{\mu}$ and $\bar{\boldsymbol{m}}^{\mu} = \lim_{n\to\infty} \frac{1}{n} \sum_{t=0}^{n-1} (\boldsymbol{m}^{\mu})^t$ are the transition probability matrix and the *limiting matrix* of the embedded Markov chain for policy μ respectively.[9]

The Bellman equations for the average-adjusted value function H^{μ} and the average-adjusted action-value function L^{μ} can be written as

$$H^{\mu}(s) = r(s,\mu(s)) - g^{\mu} y(s,\mu(s)) + \sum_{s'\in\mathcal{S}, N\in\mathbb{N}} P(s',N|s,\mu(s)) H^{\mu}(s')$$

$$L^{\mu}(s,a) = r(s,a) - g^{\mu} y(s,a) + \sum_{s'\in\mathcal{S}, N\in\mathbb{N}} P(s',N|s,a) L^{\mu}(s',\mu(s'))$$

2.3.3 Solution Methods for SMDPs

Almost all the standard solution methods for MDPs generalize easily to SMDPs. Revised policy and value iteration algorithms are straightforward, using the SMDP Bellman equations but with all other elements remaining the same. It can be shown that these algorithms converge (Howard, 1971; Puterman, 1994).

Online algorithms such as SARSA and Q-learning also generalize to the SMDP case (Bradtke and Duff, 1995) . Parr (1998) showed that the following version of Q-learning

[8]Under the *unichain* assumption, $\bar{\boldsymbol{m}}$ has equal rows. Therefore, the right hand side of Equation 2.9 is a vector with elements all equal to g^{μ}.

[9]The *limiting matrix* $\bar{\boldsymbol{m}}$ satisfies the equality $\boldsymbol{m}\bar{\boldsymbol{m}} = \bar{\boldsymbol{m}}$.

converges in the SMDP case with several small differences in the conditions and assumptions of the proof.

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha \left[r(s,a) + \gamma^N \max_{a' \in \mathcal{A}_{s'}} Q(s',a') \right]$$

This is the update formula when the agent takes action a in state s, transitions to state s', the transition takes N time steps, and the agent receives reward $r(s,a)$ on its way to state s'.

2.4 Hierarchy and Temporal Abstraction

Reasoning and learning about temporally extended actions has been studied extensively in several fields including classical AI, control theory, and RL. In this section, we look at the historical development of hierarchy and temporal abstraction in classical AI, control, and RL.

2.4.1 Temporal Abstraction in Classical AI

The problem of using abstraction to facilitate planning has been a key focus of AI research since its early days. The key idea was to replace the low-level actions available to solve a given task by **macro operators**, open-loop sequences of actions that can achieve some subgoal. It can provide exponential reduction in the computational cost of finding good plans.

Different forms of representation have been used for macro-operators, such as procedural nets (Sacerdoti, 1974), and hierarchical task networks (Currie and Tate, 1991). All these representations have these issues in common, the way in which the macro-operator selects actions, and the model it uses to predict its consequences. However, the key issue is learning useful macro-operators, which can be reused to solve different planning problems. Korf (1985) introduced a method which decomposes a planning problem to a set of independent and serializable subgoals, solves subgoals individually, and then combines the

corresponding macro-operators to solve the larger planning problems. The SOAR system (Laird et al., 1986) used a chunking mechanism, by which action sequences used to solve subtasks were memorized as macro-operators. Knoblock (1990) addressed the learning of macro-operators with the pre-conditions under which they succeed or fail. His work identifies conditions under which a solution obtained in an abstracted state and action space can be indeed executed. Drescher (1991) advocated a constructive approach in which knowledge about the world is gradually acquired in the form of **schemas**, elementary models containing a context (state), an action, and a result (new state). Schemas are built with the purpose of capturing regularities in the environment, and subsequently are used to construct new composite actions by sequencing existing primitives.

More recent research even takes into account the assumption of stochastic environment in which the plans have to be executed (Oates and Cohen, 1996; Brafman and Tennenholtz, 1997). Probabilistic and statistical methods such as belief and value function, as well as closed-loop behaviors are used to deal with such environments.

2.4.2 Temporal Abstraction in Control

Modeling and control of multiple time scale systems is an active research area in control theory where temporally extended actions and models have been extensively used. Multiple scale systems are often characterized by a fast motion superimposed over a slow motion. If the two motions do not influence each other, then the fast motion can be modeled and then eliminated to analyze the slow motion.

Perhaps the first application of temporal abstraction in stochastic control is the work by Forestier and Varaiya (1978). They proposed using a two layer system where a supervisor at the higher layer monitors the plant and intervenes only when the plant reaches a predefined boundary condition, and lower-level controls the plant between the boundary conditions. The problem of choosing the optimal lower-level controller at each boundary state is a

decision problem operating at a slower time scale with only the boundary states as states and only the lower-level controllers as actions.

The problem of controlling a system at multiple time scales has also been addressed by singular perturbation methods (Kokotovic et al., 1986; Ho and Cao, 1991; Cao et al., 2002). These methods assume that the system to be controlled has state variables with fast and slow variations. Each type of variation is modeled separately which leads to a form of hierarchical control. The slow variation states are ignored initially, and are controlled only after the fast variation states have been accounted for.

2.4.3 Temporal Abstraction in Reinforcement Learning

Temporally extended actions have been studied in hierarchical probabilistic planning and **hierarchical reinforcement learning** (HRL). HRL is a general framework for scaling RL to problems with large state spaces by using the task (or action) structure to restrict the space of policies. The key principle underlying HRL is to develop learning algorithms that do not need to learn policies from scratch, but instead reuse existing policies for simpler subtasks (or macro-actions). Macros form the basis of hierarchical specifications of action sequences because macros can include other macros in their definitions. It is similar to the familiar idea of subroutine from programming languages. A subroutine can call other subroutines as well as execute primitive commands. Most of the existing HRL models have roughly the same semantics as hierarchies of macros. However, a macro as an open-loop control policy is inappropriate for most interesting control purposes, especially the control of stochastic systems. HRL methods generalize the macro idea to closed-loop policies or more precisely, closed-loop partial policies because they are generally defined for a subset of the state space. The partial policies must also have well-defined termination conditions. These partial policies with well-defined termination conditions are sometimes called **temporally extended actions**. Work in HRL has followed three main trends: focusing on subsets of the state space in a divide-and conquer approach (state space decomposition),

grouping sequences or sets of actions together (temporal abstraction), and ignoring differences between states based on the context (state abstraction). Much of the work falls into several of these categories.

Singh (1992) introduced hierarchies of abstract actions, which achieve different tasks, as well as a hierarchy of models with variable temporal resolution. Singh used a special purpose gating architecture to switch between abstract actions, and specialized learning algorithms for this architecture. Kaelbling (1993a,b) proposed the idea of using subgoals both in order to learn sub-policies and to collapse the state space. Dayan and Hinton (1993) presented Feudal RL, a hierarchical technique which uses both temporal abstraction and state abstraction. It recursively partitions the state space and the time scale from one level to the next.

The difficulty with using the above methods was that decisions in HRL are no longer made at synchronous time steps, as is traditionally assumed in RL. Instead, agent makes decision in epochs of variable length, such as when a distinguishing state is reached (e.g., an intersection in a robot navigation task), or a subtask is completed (e.g., the elevator arrives on the first floor). Fortunately, a well-known statistical model is available to treat variable length actions: the SMDP model described in Section 2.3. Here, state transition dynamics is specified not only by the state where an action was taken, but also by parameters specifying the length of time since the action was taken. Early work in RL on the SMDP model studied extensions of algorithms such as Q-learning to continuous-time (Bradtke and Duff, 1995; Mahadevan et al., 1997b). The early work on SMDP model was then expanded to include hierarchical task models over fully or partially specified lower level subtasks, which led to developing powerful HRL models such as **hierarchies of abstract machines** (HAMs) (Parr, 1998), **options** (Sutton et al., 1999; Precup, 2000), **MAXQ** (Dietterich, 2000), and **programmable HAMs** (PHAMs) (Andre and Russell, 2001; Andre, 2003). In the options model (at least in its simplest form), Sutton et. al. studied how to learn policies given fully specified policies for executing subtasks. In the HAMs formulation, Parr

showed how hierarchical learning could be achieved even when the policies for lower-level subtasks were only partially specified. The MAXQ model is one of the first methods to combine temporal abstraction with state abstraction. It provides a more comprehensive framework for hierarchical learning where instead of policies for subtasks, the learner is given **pseudo-reward** functions. Unlike options and HAMs, MAXQ does not rely directly on reducing the entire problem to a single SMDP. Instead, a hierarchy of SMDPs is created whose solutions can be learned simultaneously. The key feature of MAXQ is the decomposed representation of the value function. Dietterich views each subtask as a separate MDP, and thus represents the value of a state within that MDP as composed of the reward for taking an action at that state (which might be composed of many rewards along a trajectory through a subtask) and the expected reward for completing the subtask. To isolate the subtask from the calling context, Dietterich uses the notion of a pseudo-reward. At the terminal states of a subtask, the agent is rewarded according to the pseudo-reward, which is set a priori by the designer, and does not depend on what happens after leaving the current subtask. Each subtask can then be treated in isolation from the rest of the problem with the caveat that the solutions learned are only recursively optimal. Each action in the recursively optimal policy is optimal with respect to the subtask containing the action, all descendant subtasks, and the pseudo-reward chosen by the designer of the system. Another important contribution of Dietterich's work is the idea that state abstraction can be done separately on the different components of the value function, which allows one to perform more abstraction. We investigate the MAXQ framework and its related concepts such as pseudo-reward, recursive optimality, value function decomposition, and state abstraction in more details in Chapter 3. In the PHAMs model, Andre and Russell extended HAMs and presented an agent design language for RL. Andre and Russell (2002) also addressed the issue of safe state abstraction in HRL. Their method yields state abstraction while maintaining hierarchical optimality.

HRL has also been successfully applied to behavior-based robotics (Brooks, 1986) in several applications (Mahadevan and Connell, 1992; Lin, 1993; Digney, 1996; Mataric, 1997; Huber and Grupen, 1997). Mahadevan and Connell used a subsumption architecture in which simple behaviors are acquired using RL and then are combined by a pre-defined scheme to solve a complex robot box-pushing task. Lin used the decomposition of a complex task into smaller subtasks, each having its own limited state space and its own reward function. A robot can learn a behavior for solving each subtask, and then use RL at the higher level in order to determine the best combination of sub-behaviors. Huber used RL and a hybrid discrete event dynamical system to learn walking gaits for a robot. At the low level, the robot uses a set of pre-existing controllers that can generate collision-free motion and optimize forces and posture. At the higher level, RL is used to determine which controller should be applied, depending on a set of discrete variables describing the state of the system.

Recent research is also targeted toward finding temporally extended actions automatically. Thrun and Schwartz (1995) and Pickett and Barto (2002) generate temporal abstractions by finding commonly occurring sub-policies in solutions to a set of tasks. Digney (1996), McGovern and Barto (2001), Menache et al. (2002), and Simsek and Barto (2004) identify subgoal states and generate temporally extended actions that take the agent to these states. Digney's subgoals are states that are visited frequently or that have a high reward gradient. McGovern and Barto's method identifies as subgoals those regions of the state space that the agent visits frequently on successful trajectories but not on unsuccessful ones. Menache et al. define subgoals as the border states of strongly connected areas of the MDP transition graph and find them using a max-flow/min-cut algorithm. Simsek and Barto propose a method to identify useful temporal abstractions using relative novelty. Their definition of novelty relates it to how frequently a state is visited since a designated start time. They define relative novelty of a state in a transition sequence as the ratio of the novelty of states that followed it (including itself) to the novelty of the states that pre-

ceded it. Hengst (2002) and Jonsson and Barto (2005) proposed constructing a hierarchy of abstractions in problems with factored state spaces. Hengst's method orders state variables with respect to their frequency of change and adds a layer of hierarchy for each state variable, where each layer handles a smaller MDP than its lower layers. Jonsson and Barto determine causal relationships between state variables using a dynamic Bayesian network (DBN) model of factored MDPs and like Hengst's algorithm, their algorithm introduces layers of temporally extended actions based on the causal structure of the task. Mannor et al. (2004) find clusters of states and define temporally extended actions as a sub-policy that allows the agent to efficiently shift from one cluster to the other. They use two different clustering mechanisms, one that employs only topology, and one that uses the reward structure of the problem in addition to topology.

2.5 Multi-Agent Reinforcement Learning

The analysis of multi-agent systems is a topic of interest in both economic theory and AI. Their integration with existing methods in AI constitutes a promising area of research. An optimal policy in a multi-agent system may depend on the behavior of other agents, which is often not predictable. It makes learning and adaptation a necessary component of an agent. Multi-agent learning studies algorithms for selecting actions for multiple agents coexisting in the same environment. This is a complicated problem, because the behaviors of the other agents can be changing as they also adapt to achieve their own goals. It usually makes the environment non-stationary and often non-Markovian as well (Mataric, 1997). Robosoccer; disaster rescue, where robots must safely find victims as fast as possible after an earthquake; e-commerce; manufacturing systems, where managers of a factory coordinate to maximize the profit; and distributed sensor networks, where multiple sensors collaborate to perform a large-scale sensing task under strict power constraints, are examples of challenging multi-agent domains that need robust learning algorithms for co-

ordination among multiple agents or effectively responding to other agents (Weiss, 1999; Lesser et al., 2003).

In addition to the existing methods in distributed AI and machine learning, game theory also provides a framework for research in multi-agent learning. The game theoretic concepts of **stochastic game** and **Nash equilibria** (Owen, 1995; Filar and Vrieze, 1997) are the foundation for much of the recent research in multi-agent learning. Learning algorithms use stochastic games as a natural extension of MDPs to multiple agents. These algorithms can be summarized by broadly grouping them into two categories: **equilibria learners** and **best-response learners**. Equilibria learners such as Minimax-Q (Littman, 1994), Nash-Q (Hu and Wellman, 1998), the gradient ascent learner in (Singh et al., 2000b), and Friend-or-Foe-Q (Littman, 2001) seek to learn an equilibrium of the game by iteratively computing intermediate equilibria. They guarantee convergence to their part of an equilibrium solution regardless of the behavior of the other agents. On the other hand, best-response learners seek to learn the best response to the other agents. Although not an explicitly multi-agent algorithm, Q-learning (Watkins, 1989) was one of the first algorithms applied to multi-agent problems (Tan, 1993; Crites and Barto, 1998). Joint-state/joint-action learners (Boutilier, 1999) and WoLF-PHC (Bowling and Veloso, 2002) are another examples of a best-response learner. It has been shown by Bowling and Veloso (2002) that if an algorithm in which best-response learners playing with each other converges, it must be to a Nash equilibrium.

Multi-agent learning has been recognized to be challenging for two main reasons: **1) curse of dimensionality**: the number of parameters to be learned increases dramatically with the number of agents, and **2) partial observability**: states and actions of the other agents which are required for an agent to make decision are not fully observable and inter-agent communication is usually costly.

Prior work in multi-agent learning have addressed the curse of dimensionality in many different ways. One natural approach is to restrict the amount of information that is avail-

able to each agent and hope to maximize the global payoff by solving local optimization problems for each agent. This idea has been addressed using value function RL (Schneider et al., 1999) as well as policy gradient RL (Peshkin et al., 2000). Another approach is to exploit the structure in a multi-agent problem using factored value functions. Guestrin et al. (2002) integrate these ideas in collaborative multi-agent domains. They use value function approximation and approximate the joint value function as a linear combination of local value functions, each of which relates only to the parts of the system controlled by a small number of agents. Factored value functions allow the agents to find a globally optimal joint-action using a message passing scheme. However, this approach does not address the communication cost in its message passing strategy.

Graphical models have also been used to address the curse of dimensionality in multi-agent systems. This work seeks to transfer the representational and computational benefits that graphical models provide to probabilistic inference in multi-agent systems and game theory (La-Mura, 2000; Koller and Milch, 2001). The previous work established algorithms for computing Nash equilibria in one-stage games, including efficient algorithms for computing approximate (Kearns et al., 2001) and exact (Littman et al., 2002) Nash equilibria in tree-structured games, and convergent heuristics for computing Nash equilibria in general graphs (Vickrey and Koller, 2002; Ortiz and Kearns, 2003).

The curse of dimensionality has also been addressed in multi-agent robotics. Multi-robot learning methods usually reduce the complexity of the problem by not modeling joint states or actions explicitly, such as work by Mataric (1997) and Balch and Arkin (1998), among others. In such systems, each robot maintains its position in a formation depending on the locations of the other robots, so there is some implicit communication or sensing of states and actions of the other agents. There has also been work on reducing the parameters needed for Q-learning in multi-agent domains by learning action-values over a set of derived features (Stone and Veloso, 1999). These derived features are domain specific and have to be encoded by hand, or constructed by a supervised learning algorithm.

Almost all the above methods ignore the problem that an agent might not have free access to the other agents' information that are required to make its own decision. In general, the world is partially observable for each agent in a distributed multi-agent setting. POMDPs have been used to model partial observability in probabilistic AI. The POMDP framework can be extended to allow for multiple distributed agents to base their decisions on their local observations. This model is called **decentralized POMDP** (DEC-POMDP) and it has been shown that the decision problem for a DEC-POMDP is NEXP-complete (Bernstein et al., 2000). One way to address partial observability in distributed multi-agent domains is to use communication to exchange required information. However, since communication can be costly, in addition to its normal actions, each agent needs to decide about communication with other agents (Xuan et al., 2001; Xuan and Lesser, 2002). Pynadath and Tambe (2002) extended DEC-POMDP by including communication decisions in the model, and proposed a framework called **communicative multi-agent team decision problem** (COM-MTDP). Since DEC-POMDP can be reduced to COM-MTDP with no communication by copying all the other model features, decision problem for a COM-MTDP is also NEXP-complete (Pynadath and Tambe, 2002). The trade-off between the quality of solution, the cost of communication, and the complexity of the model is currently a very active area of research in multi-agent learning and planning.

CHAPTER 3

A FRAMEWORK FOR HIERARCHICAL REINFORCEMENT LEARNING

In this chapter, we introduce a general hierarchical reinforcement learning (HRL) framework for simultaneous learning of policies at multiple levels of hierarchy. Our treatment builds upon the existing approaches such as HAMs (Parr, 1998), options (Sutton et al., 1999; Precup, 2000), MAXQ (Dietterich, 2000), and PHAMs (Andre and Russell, 2002; Andre, 2003), especially the MAXQ value function decomposition. In our framework, we add three-part value function decomposition (Andre and Russell, 2002) to guarantee hierarchical optimality, and reward shaping (Ng et al., 1999) to reduce the burden of exploration, to the MAXQ method. Rather than redundantly explain MAXQ and then our hierarchical framework, we will present our model and note throughout this chapter where the key pieces were inspired by or are directly related to Dietterich's MAXQ work. In the following chapters, we first extend this framework to the average reward model, then we generalize it to be applicable to problems with continuous state and/or action spaces, and finally broaden it to be appropriate for domains with multiple cooperative agents.

3.1 Motivating Example

In the HRL framework, the designer of the system imposes a hierarchy on the problem to incorporate domain knowledge and thereby reduces the size of the space that must be searched to find a good policy. The designer recursively decomposes the overall task into a collection of subtasks that she/he believes are important for solving the problem.

Let us illustrate the main ideas using a simple search task shown in Figure 3.1. Consider the case where, in an office (rooms and connecting corridors) type environment, a robot is

assigned the task of picking up trash from trash cans ($T1$ and $T2$) over an extended area and accumulating it into one centralized trash bin ($Dump$), from where it might be sent for recycling or disposed. For simplicity, we assume that the robot can observe its true location in the environment. The main subtasks in this problem are *root* (the whole trash collection task), *collect trash at* $T1$ and $T2$, *navigate to* $T1$, $T2$, and $Dump$. Each of these subtasks is defined by a set of termination states. After defining subtasks, we must indicate for each subtask, which other subtasks or primitive actions it should employ to reach its goal. For example, *navigate to* $T1$, $T2$, and $Dump$ use three primitive actions *find wall*, *align with wall*, and *follow wall*. *Collect trash at* $T1$ uses two subtasks *navigate to* $T1$ and $Dump$, plus two primitive actions *Put* and *Pick*, and so on. Like MAXQ, all of this information can be summarized by a directed acyclic graph called the **task graph**. The task graph for the trash collection problem is shown in Figure 3.1. This hierarchical model is able to support **state abstraction** (while the agent is moving toward the $Dump$, the status of trash cans $T1$ and $T2$ is irrelevant and cannot affect this navigation process. Therefore, the variables defining the status of trash cans $T1$ and $T2$ can be removed from the state space of the *navigate to Dump* subtask) and **subtask sharing** (if the system could learn how to solve the *navigate to* $Dump$ subtask once, then the solution could be shared by both *collect trash at* $T1$ and $T2$ subtasks).

Like HAMs (Parr, 1998), options (Sutton et al., 1999; Precup, 2000), MAXQ (Dietterich, 2000), and PHAMs (Andre and Russell, 2001; Andre, 2003), this framework also relies on the theory of SMDPs. While SMDP theory provides the theoretical underpinnings of temporal abstraction by modeling actions that take varying amounts of time, the SMDP model provides little in the way of concrete representational guidance, which is critical from a computational point of view. In particular, the SMDP model does not specify how tasks can be broken up into subtasks, how to decompose value functions, etc. We examine these issues next.

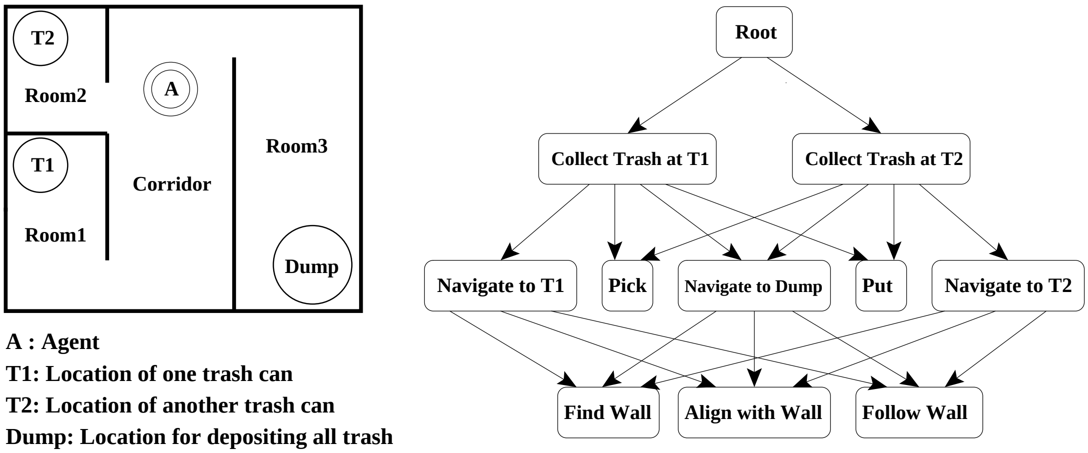


Figure 3.1. A robot trash collection task and its associated task graph.

As in MAXQ, a task hierarchy such as the one illustrated above can be modeled by decomposing the overall task MDP $\mathcal{M}$, into a finite set of subtasks $\{M_0, M_1, \ldots, M_{m-1}\}$, where M_0 is the *root* task. Solving M_0 solves the entire MDP $\mathcal{M}$.

Definition 3.1: Each **non-primitive** subtask M_i (M_i is not a primitive action) consists of five components $(S_i, \mathcal{I}_i, T_i, A_i, R_i)$:

- S_i is the **state space** for subtask M_i. It is described by those state variables that are relevant to subtask M_i. The range of a state variable describing S_i might be a subset of its range in $\mathcal{S}$ (the state space of MDP $\mathcal{M}$).

- $\mathcal{I}_i \subseteq S_i$ is the **initiation set** for subtask M_i. Subtask M_i can be initiated only in states belonging to $\mathcal{I}_i$.

- $T_i \subseteq S_i$ is the **set of terminal states** for subtask M_i. Subtask M_i terminates when it reaches a state in T_i. A policy for subtask M_i can only be executed if the current state s belongs to $(S_i - T_i)$.

- A_i is the **set of actions** that can be performed to achieve subtask M_i. These actions can be either primitive actions from $\mathcal{A}$ (the set of primitive actions for MDP $\mathcal{M}$),

or they can be other subtasks. Technically, A_i is a function of states, since it may differ from one state to another. However, we will suppress this dependence in our notation.

- R_i is the **reward structure** inside subtask M_i and could be different from the reward function of MDP $\mathcal{M}$. Here we use the idea of reward shaping (Ng et al., 1999) and define a more general reward structure than MAXQ's, which specifies a pseudo-reward only for transitions to terminal states. Reward shaping is a method for guiding an agent toward a solution without constraining the search space. Besides the reward of the overall task MDP $\mathcal{M}$, each subtask M_i can use additional rewards to guide its local learning. Additional rewards are only used inside each subtask and do not propagate to upper levels in the hierarchy. If the reward structure inside a subtask is different from the reward function of the overall task, we need to define two types of value functions for each subtask, internal value function and external value function. The **internal value function** is defined based on both the local reward structure of the subtask and the reward of the overall task, and only used in learning the subtask. On the other hand, the **external value function** is defined only based on the reward function of the overall task and is propagated to the higher levels in the hierarchy to be used in learning the global policy. This reward structure for each subtask in our framework is more general than the one in MAXQ, and of course, includes the MAXQ's pseudo-reward.[1] $\square$

Each primitive action a is a primitive subtask in this decomposition, such that a is always executable and it terminates immediately after execution. From now on in this thesis, we use subtask to refer to non-primitive subtasks.

[1]The MAXQ pseudo-reward function is defined only for transitions to terminal states, and is zero for non-terminal states.

3.2 Policy Execution

If we have a policy for each subtask in the hierarchy, we can define a **hierarchical policy** for the model.

Definition 3.2: A hierarchical policy $\boldsymbol{\mu}$ is a set of policies, one policy for each of the subtasks in the hierarchy: $\boldsymbol{\mu} = \{\mu_0, \ldots, \mu_{m-1}\}$. □

The hierarchical policy is executed using a stack discipline, similar to ordinary programming languages. Each subtask policy takes a state and returns the name of a primitive action to execute or the name of a subtask to invoke. When a subtask is invoked, its name is pushed onto the **Task-Stack** and its policy is executed until it enters one of its terminal states. When a subtask terminates, its name is popped off the Task-Stack. If any subtask on the Task-Stack terminates, then all subtasks below it are immediately aborted, and control returns to the subtask that had invoked the terminated subtask. Hence, at any time, the *root* task is located at the bottom and the subtask which is currently being executed is located at the top of the Task-Stack.

Under a hierarchical policy $\boldsymbol{\mu}$, we define a multi-step transition probability $P_i^{\boldsymbol{\mu}} : S_i \times \mathbb{N} \times S_i \rightarrow [0, 1]$ for each subtask M_i in the hierarchy, where $P_i^{\boldsymbol{\mu}}(s', N|s)$ denotes the probability that hierarchical policy $\boldsymbol{\mu}$ will cause the system to transition from state s to state s' in N time steps at subtask M_i. We also define a multi-step abstract transition probability $F_i^{\boldsymbol{\mu}} : S_i \times \mathbb{N} \times S_i \rightarrow [0, 1]$ for each subtask M_i under the hierarchical policy $\boldsymbol{\mu}$. The term $F_i^{\boldsymbol{\mu}}(s', N|s)$ denotes the N-step abstract transition probability from state s to state s' under hierarchical policy $\boldsymbol{\mu}$ at subtask M_i, where N is the number of actions taken by subtask M_i, not the number of primitive actions taken in this transition. In this thesis, we use the multi-step abstract transition probability $F_i^{\boldsymbol{\mu}}$ to model state transition at the subtask level, and the multi-step transition probability $P_i^{\boldsymbol{\mu}}$ to model state transition at the level of primitive actions. Finally, we define a single-step transition probability $P^{\boldsymbol{\mu}} : \mathcal{S} \times \mathcal{S} \rightarrow [0, 1]$

under the hierarchical policy $\boldsymbol{\mu}$, where $P^{\boldsymbol{\mu}}(s'|s)$ denotes the probability that the hierarchical policy $\boldsymbol{\mu}$ will cause the system to transition from state s to state s' at the level of primitive actions.

3.3 Local versus Global Optimality

Using hierarchy reduces the size of the space that must be searched to find a good policy. However, a hierarchy constrains the space of possible policies so that it may not be possible to represent the optimal policy or its value function, and hence make it impossible to learn the optimal policy. If we cannot learn the optimal policy, the next best target would be to learn the best policy that is consistent with the given hierarchy. Two notions of optimality have been explored in the previous work on hierarchical reinforcement learning, **hierarchical optimality** and **recursive optimality** (Dietterich, 2000).

Definition 3.3: A hierarchical optimal policy for MDP $\mathcal{M}$ is a hierarchical policy which has the best performance among all policies consistent with the given hierarchy. In other words, hierarchical optimality is a global optimum consistent with the given hierarchy. In this form of optimality, the policy for each individual subtask is not necessarily optimal, but the policy for the entire hierarchy is optimal. The HAMQ HRL algorithm (Parr, 1998) and the SMDP Q-learning algorithm for a fixed set of options (Sutton et al., 1999; Precup, 2000) both converge to a hierarchically optimal policy. $\square$

Definition 3.4: Recursive optimality, first introduced by Dietterich (2000), is a weaker but more flexible form of optimality which only guarantees that the policy of each subtask is optimal given the policies of its children. It is an important and flexible form of optimality because it permits each subtask to learn a locally optimal policy while ignoring the behavior of its ancestors in the hierarchy. This increases the opportunity for subtask

sharing and state abstraction. The MAXQ-Q HRL algorithm (Dietterich, 2000) converges to a recursively optimal policy. □

3.4 Value Function Definitions

For recursive optimality, the goal is to find a hierarchical policy $\boldsymbol{\mu} = \{\mu_0, \dots, \mu_{m-1}\}$ such that for each subtask M_i in the hierarchy, the expected cumulative reward of executing policy μ_i and the policies of all descendants of M_i is maximized. In this case, the value function to be learned for subtask M_i under hierarchical policy $\boldsymbol{\mu}$ must contain only the reward received during the execution of subtask M_i. We call this the **projected value function** after Dietterich (2000), and define it as follows:

Definition 3.5: The projected value function of a hierarchical policy $\boldsymbol{\mu}$ on subtask M_i, denoted $\hat{V}^{\boldsymbol{\mu}}(i, s)$, is the expected cumulative reward of executing policy μ_i and the policies of all descendants of M_i starting in state $s \in S_i$ until M_i terminates. □

The expected cumulative reward outside a subtask is not a part of its projected value function. It makes the projected value function of a subtask dependent only on the subtask and its descendants.

On the other hand, for hierarchical optimality, the goal is to find a hierarchical policy that maximizes the expected cumulative reward. In this case, the value function to be learned for subtask M_i under hierarchical policy $\boldsymbol{\mu}$ must contain the reward received during the execution of subtask M_i, and the reward after subtask M_i terminates. We call this the **hierarchical value function** following Dietterich (2000). The hierarchical value function of a subtask includes the expected reward outside the subtask and therefore depends on the subtask and all its ancestors up to the root of the hierarchy. In the case of hierarchical optimality, we need to consider the contents of the Task-Stack as an additional part of the state space of the problem, since a subtask might be shared by multiple parents.

Definition 3.6: Ω is the space of possible values of the Task-Stack for hierarchy $\mathcal{H}$. $\square$

Let us define joint state space $\mathcal{X} = \Omega \times \mathcal{S}$ for the hierarchy $\mathcal{H}$ as the cross product of the set of the Task-Stack values Ω and the states space $\mathcal{S}$. We define the hierarchical value function using joint state space $\mathcal{X}$ as

Definition 3.7: A hierarchical value function for subtask M_i in state $x = (\omega, s)$ under hierarchical policy $\boldsymbol{\mu}$, denoted $V^{\boldsymbol{\mu}}(i, x)$, is the expected cumulative reward of following the hierarchical policy $\boldsymbol{\mu}$ starting in state $s \in S_i$ and Task-Stack ω. $\square$

The current subtask M_i is a part of the Task-Stack ω and as a result is a part of the state x. So we can exclude it from the hierarchical value function notation and write $V^{\boldsymbol{\mu}}(i, x)$ as $V^{\boldsymbol{\mu}}(x)$. However for clearance, we use $V^{\boldsymbol{\mu}}(i, x)$ in the rest of this dissertation.

Theorem 3.1: Under a hierarchical policy $\boldsymbol{\mu}$, each subtask M_i can be modeled by an SMDP consisting of components $(S_i, A_i, P_i^{\boldsymbol{\mu}}, \bar{R}_i)$, where $\forall a \in A_i, \bar{R}_i(s, a) = \hat{V}^{\boldsymbol{\mu}}(a, s)$. $\square$

This theorem is similar to Theorem 1 in Dietterich (2000). Using this theorem, we can define a recursive optimal policy for MDP $\mathcal{M}$ with hierarchical decomposition $\{M_0, M_1, \ldots, M_{m-1}\}$ as a hierarchical policy $\boldsymbol{\mu} = \{\mu_0, \ldots, \mu_{m-1}\}$ such that for each subtask M_i, the corresponding policy μ_i is optimal for the SMDP defined by the tuple $(S_i, A_i, P_i^{\boldsymbol{\mu}}, \bar{R}_i)$.

3.5 Value Function Decomposition

A value function decomposition splits the value of a state or a state-action pair into multiple additive components. Modularity in the hierarchical structure of a task allows us to carry out this decomposition along subtask boundaries. In this section, we first describe

the two-part or MAXQ decomposition proposed by Dietterich (2000), and then the three-part decomposition proposed by Andre and Russell (2002). We use both decompositions in our hierarchical framework depending on the type of optimality (hierarchical or recursive) that we are interested in.

The two-part value function decomposition is at the center of the MAXQ method. The purpose of this decomposition is to decompose the projected value function of the *root* task, $\hat{V}^{\boldsymbol{\mu}}(0,s)$, in terms of the projected value functions of all of the subtasks in the hierarchy. The projected value of subtask M_i at state s under hierarchical policy $\boldsymbol{\mu}$ can be written as

$$\hat{V}^{\boldsymbol{\mu}}(i,s) = E\left[\sum_{k=0}^{\infty} \gamma^k r(s_k,a_k)|s_0 = s, \boldsymbol{\mu}\right] \tag{3.1}$$

Now let us suppose that the first action chosen by μ_i is invoked and it executes for a number of steps N and terminates in state s' according to $P_i^{\boldsymbol{\mu}}(s',N|s)$. We can re-write Equation 3.1 as

$$\hat{V}^{\boldsymbol{\mu}}(i,s) = E\left[\sum_{k=0}^{N-1} \gamma^k r(s_k,a_k) + \sum_{k=N}^{\infty} \gamma^k r(s_k,a_k)|s_0 = s, \boldsymbol{\mu}\right] \tag{3.2}$$

The first summation on the right-hand side of Equation 3.2 is the discounted sum of rewards for executing subtask $\mu_i(s)$ starting in state s until it terminates, in other words, it is $\hat{V}^{\boldsymbol{\mu}}(\mu_i(s),s)$, the projected value function of the child task $\mu_i(s)$. The second term on the right-hand side of the equation is the projected value of state s' for the current task M_i, $\hat{V}^{\boldsymbol{\mu}}(i,s')$, discounted by γ^N, where s' is the current state when subroutine $\mu_i(s)$ terminates and N is the number of transition steps from state s to state s'. We can therefore write Equation 3.2 in the form of a Bellman equation:

$$\hat{V}^{\boldsymbol{\mu}}(i,s) = \hat{V}^{\boldsymbol{\mu}}(\mu_i(s),s) + \sum_{s',N} P_i^{\boldsymbol{\mu}}(s',N|s)\gamma^N \hat{V}^{\boldsymbol{\mu}}(i,s') \tag{3.3}$$

Equation 3.3 can be re-stated for the projected action-value function as follows:

$$\hat{Q}^{\boldsymbol{\mu}}(i,s,a) = \hat{V}^{\boldsymbol{\mu}}(a,s) + \sum_{s',N} P_i^{\boldsymbol{\mu}}(s',N|s,a)\gamma^N \hat{Q}^{\boldsymbol{\mu}}(i,s',\mu_i(s')) \tag{3.4}$$

The right-most term in this equation is the expected discounted cumulative reward of completing subtask M_i after executing subtask M_a in state s. Dietterich called this term **completion function** and is denoted by $C^{\boldsymbol{\mu}}(i,s,a)$. With this definition, we can express the projected action-value function recursively as

$$\hat{Q}^{\boldsymbol{\mu}}(i,s,a) = \hat{V}^{\boldsymbol{\mu}}(a,s) + C^{\boldsymbol{\mu}}(i,s,a) \tag{3.5}$$

and we can re-express the definition for projected value function as

$$\hat{V}^{\boldsymbol{\mu}}(i,s) = \begin{cases} \hat{Q}^{\boldsymbol{\mu}}(i,s,\mu_i(s)) & \text{if } M_i \text{ is a non-primitive subtask,} \\ \sum_{s'} P(s'|s,i)r(s,i) & \text{if } M_i \text{ is a primitive action.} \end{cases} \tag{3.6}$$

Equations 3.5 and 3.6 are referred to as two-part decomposition equations for a hierarchy under a fixed hierarchical policy $\boldsymbol{\mu}$. These equations recursively decompose the projected value function for the *root* into the projected value functions for the individual subtasks, $M_1, \ldots, M_{m-1}$, and the individual completion functions $C^{\boldsymbol{\mu}}(j,s,a)$ for $j = 1, \ldots, m-1$. The fundamental quantities that must be stored to represent the value function decomposition are the C values for all non-primitive subtasks and the V values for all primitive actions.[2] The two-part decomposition is summarized graphically in Figure 3.2. As mentioned in Section 3.4, since the expected reward after execution of subtask M_i is not a component of the projected action-value function, the two-part decomposition allows only for recursive optimality.

[2]The projected value function and value function are the same for a primitive action.

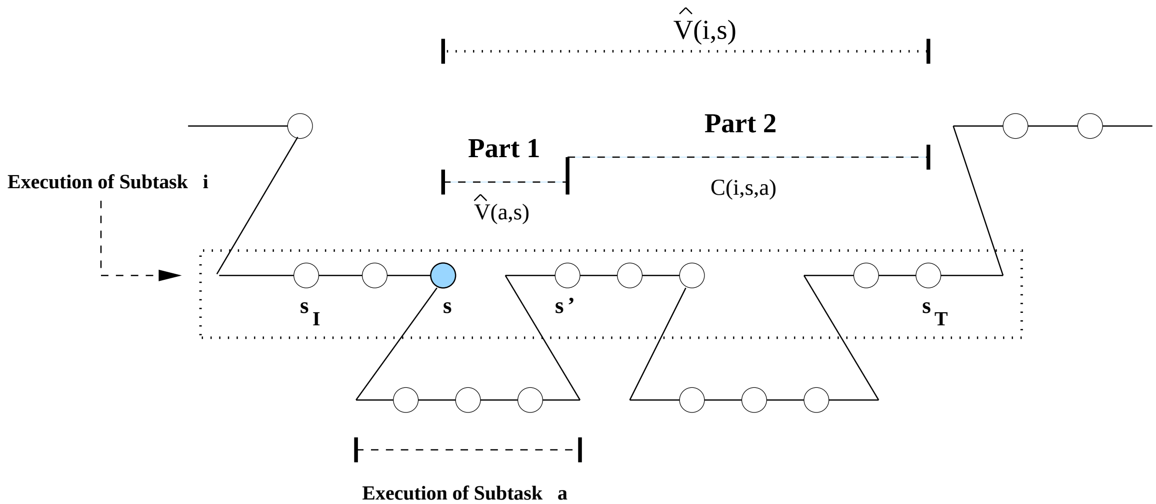


Figure 3.2. This figure shows the two-part decomposition for $\hat{V}(i, s)$, the projected value function of subtask M_i for the shaded state s. Each circle is a state of the SMDP visited by the agent. Subtask M_i is initiated at state s_I and terminates at state s_T. The projected value function $\hat{V}(i, s)$ is broken into two parts: **Part 1)** the projected value function of subtask M_a for state s, and **Part 2)** the completion function, the expected discounted cumulative reward of completing subtask M_i after executing subtask M_a in state s.

Andre and Russell (2002) proposed a three-part value function decomposition for achieving hierarchical optimality. They add a third component for the expected sum of rewards outside the current subtask to the two-part value function decomposition. This decomposition decomposes the hierarchical value function of each subtask into three parts. As shown in Figure 3.3, these three parts correspond to executing the current action (which might itself be a subtask), completing the rest of the current subtask (so far is similar to the MAXQ decomposition), and all actions outside the current subtask.

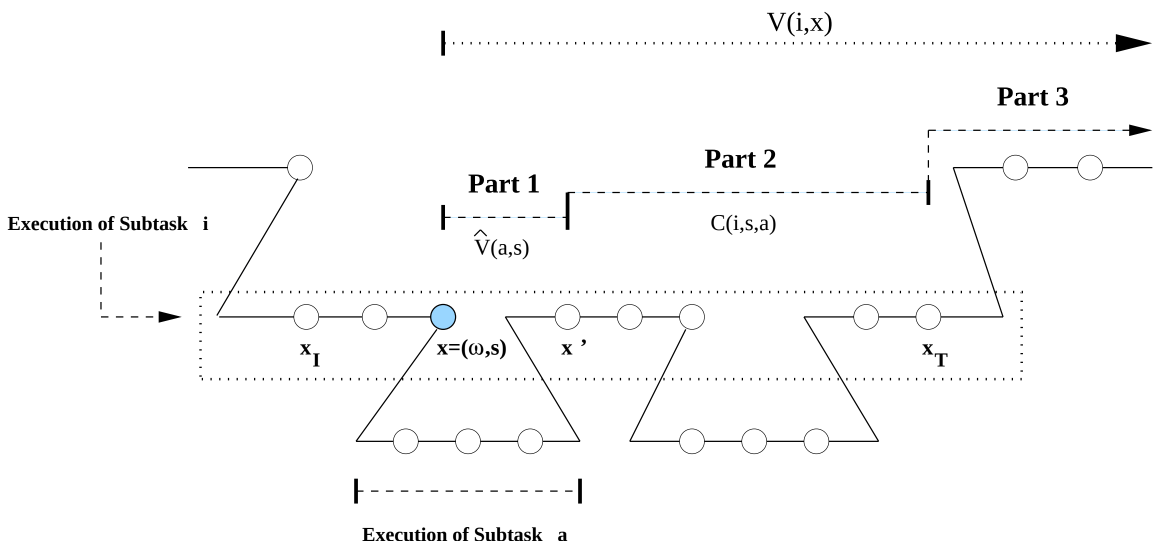


Figure 3.3. This figure shows the three-part decomposition for $V(i, x)$, the hierarchical value function of subtask M_i for the shaded state $x = (\omega, s)$. Each circle is a state of the SMDP visited by the agent. Subtask M_i is initiated at state x_I and terminates at state x_T. The hierarchical value function $V(i, x)$ is broken into three parts: **Part 1)** the projected value function of subtask M_a for state s, **Part 2)** the completion function, the expected discounted cumulative reward of completing subtask M_i after executing subtask M_a in state s, and **Part 3)** the sum of all rewards after termination of subtask M_i.

CHAPTER 4

HIERARCHICAL AVERAGE REWARD REINFORCEMENT LEARNING

As described in Chapter 2, the average-reward formulation is more appropriate for a wide class of *continuing* tasks than more well-studied discounted reward framework. A primary goal of continuing tasks, including manufacturing, scheduling, queuing, and inventory control, is to find a *gain-optimal policy* that maximizes (minimizes) the long-run average reward (cost) over time. Although average reward reinforcement learning (RL) has been studied using both the discrete-time MDP model (Schwartz, 1993; Mahadevan, 1996; Tadepalli and Ok, 1996; Marbach, 1998; Van-Roy, 1998) as well as the continuous-time SMDP model (Mahadevan et al., 1997b; Wang and Mahadevan, 1999), prior work has been limited to *flat* policy representations.

In this chapter,[1] we extend previous work on hierarchical reinforcement learning (HRL) to the average reward framework, and investigate two formulations of HRL based on the average reward SMDP model. These two formulations correspond to two notions of optimality in HRL: *hierarchical optimality* and *recursive optimality* described in Section 3.3. We present discrete-time and continuous-time algorithms that learn to find hierarchically and recursively optimal average reward policies. In these algorithms, we assume that the overall task (the *root* of the hierarchy) is continuing. In the **hierarchically optimal average reward RL** (HAR) algorithms, the aim is to find a hierarchical policy within the space of

---

[1]Most of the work presented in this chapter first appeared in **1)** Ghavamzadeh and Mahadevan (2001), "Continuous-Time Hierarchical Reinforcement Learning," Proceedings of the Eighteenth International Conference on Machine Learning", pp. 186-193, and **2)** Ghavamzadeh and Mahadevan (2002), "Hierarchically Optimal Average Reward Reinforcement Learning," Proceedings of the Nineteenth International Conference on Machine Learning", pp. 195-202.

policies defined by the hierarchical decomposition that maximizes the *global gain*. In the **recursively optimal average reward RL** (RAR) algorithms, we treat subtasks as continuing average reward problems, where the goal at each subtask is to maximize its gain given the policies of its children. We investigate the conditions under which the policy learned by the RAR algorithm at each subtask is independent of the context in which it is executed and therefore can be reused by other hierarchies. We use two experimental testbeds to study the empirical performance of the proposed algorithms. The first problem is a small automated guided vehicle (AGV) scheduling task. The second problem is a relatively large AGV scheduling task. We model the second AGV task using both discrete-time and continuous-time models. We compare the performance of our proposed algorithms with other HRL methods and a flat average reward RL algorithm in this task.

The rest of this chapter is organized as follows. In Section 4.1, we present discrete-time and continuous-time *hierarchically optimal average reward RL* (HAR) algorithms. In Section 4.2, we investigate different methods to formulate subtasks in a recursively optimal average reward RL setting, and present discrete-time and continuous-time *recursively optimal average reward RL* (RAR) algorithms. We demonstrate the type of optimality achieved by HAR and RAR algorithms as well as their performance and speed compared to other algorithms in Section 4.3. Finally, Section 4.4 summarizes the chapter and discusses some directions for future work.

4.1 Hierarchically Optimal Average Reward RL Algorithm

Given the basic concepts of the average reward MDP and the average reward SMDP models described in Sections 2.2.3 and 2.3.2, and the fundamental principles of HRL and the HRL framework illustrated in Chapter 3, we can now proceed to describe a hierarchically optimal average reward RL formulation. Since we are interested in hierarchical optimality, we include the contents of the Task-Stack as a part of the state space of the problem. In this section, we consider HRL problems for which the following assumptions

hold.

Assumption 4.1 (Continuing Root Task): The *root* of the hierarchy is a *continuing* task, i.e., the root task goes on continually without termination. □

Assumption 4.2: For every hierarchical policy $\boldsymbol{\mu}$, the single-step transition probability matrix $\boldsymbol{P}^{\boldsymbol{\mu}}$ is *unichain*, that is, it consists of a single recurrent class plus a possibly empty set of transient states. □

If Assumptions 4.1 and 4.2 hold, using Equation 2.5, the gain[2]

$$g^{\boldsymbol{\mu}} = \lim_{N \to \infty} \frac{1}{N} \sum_{t=0}^{N-1} (\boldsymbol{P}^{\boldsymbol{\mu}})^t r(x, \boldsymbol{\mu}(x)) = \bar{\boldsymbol{P}}^{\boldsymbol{\mu}} r(x, \boldsymbol{\mu}(x)) \tag{4.1}$$

is well defined for every hierarchical policy $\boldsymbol{\mu}$ and does not depend on the initial state. We call $g^{\boldsymbol{\mu}}$ the **global gain** under the hierarchical policy $\boldsymbol{\mu}$. The *global gain*, $g^{\boldsymbol{\mu}}$, is the gain of the Markov chain that will result from flattening the hierarchy using the hierarchical policy $\boldsymbol{\mu}$.

We are interested in finding a hierarchical policy $\boldsymbol{\mu}^*$ which maximizes the *global gain*, i.e.,

$$g^{\boldsymbol{\mu}^*} \geq g^{\boldsymbol{\mu}}, \qquad \text{for all } \boldsymbol{\mu} \tag{4.2}$$

We refer to a hierarchical policy $\boldsymbol{\mu}^*$ which satisfies Equation 4.2 as a *hierarchically optimal average reward* policy, and to $g^{\boldsymbol{\mu}^*}$ as the *optimal average reward* or the *optimal gain*.

[2]Under the *unichain* assumption, $\bar{\boldsymbol{P}}^{\boldsymbol{\mu}}$ has equal rows. Therefore, the right hand side of Equation 4.1 is a vector with elements all equal to $g^{\boldsymbol{\mu}}$.

Here we replace value and action-value functions in the hierarchical model of Chapter 3 with average-adjusted value and average-adjusted action-value functions described in Sections 2.2.3 and 2.3.2.

The hierarchical average-adjusted value function for hierarchical policy $\boldsymbol{\mu}$ and subtask M_i, denoted $H^{\boldsymbol{\mu}}(i,x)$, is the average-adjusted sum of rewards earned by following hierarchical policy $\boldsymbol{\mu}$ starting in state $x = (\omega, s)$ until M_i terminates, plus the expected average-adjusted reward outside subtask M_i.

$$H^{\boldsymbol{\mu}}(i,x) = \lim_{N\to\infty} E\left\{\sum_{k=0}^{N-1}\left[r(x_k,a_k) - g^{\boldsymbol{\mu}}\right]|x_0 = x, \boldsymbol{\mu}\right\} \tag{4.3}$$

Here the rewards are adjusted with $g^{\boldsymbol{\mu}}$, the *global gain* under the hierarchical policy $\boldsymbol{\mu}$.

Now let us suppose that the first action chosen by μ_i is executed for a number of primitive steps N_1 and terminates in state $x_1 = (\omega, s_1)$ according to multi-step transition probability $P_i^{\boldsymbol{\mu}}(x_1, N_1|x, \mu_i(x))$, and after that subtask M_i itself executes for N_2 steps at the level of subtask M_i (N_2 is the number of actions taken by subtask M_i, not the number of primitive actions) and terminates in state $x_2 = (\omega, s_2)$ according to multi-step abstract transition probability $F_i^{\boldsymbol{\mu}}(x_2, N_2|x_1)$. We can re-write Equation 4.3 in the form of a Bellman equation as

$$\begin{aligned} H^{\boldsymbol{\mu}}(i,x) = {} & r_i^{\boldsymbol{\mu}}(x,\mu_i(x)) - g^{\boldsymbol{\mu}} y_i^{\boldsymbol{\mu}}(x,\mu_i(x)) + \\ & \sum_{N_1,s_1\in S_i} P_i^{\boldsymbol{\mu}}(x_1,N_1|x,\mu_i(x))\left[\hat{H}^{\boldsymbol{\mu}}(i,x_1) + \sum_{N_2,s_2\in S_i} F_i^{\boldsymbol{\mu}}(x_2,N_2|x_1) H^{\boldsymbol{\mu}}(Parent(i),(\omega \nearrow i, s_2))\right] \end{aligned} \tag{4.4}$$

where $\hat{H}^{\boldsymbol{\mu}}(i,.)$ is the projected average-adjusted value function of hierarchical policy $\boldsymbol{\mu}$ and subtask M_i, $y_i^{\boldsymbol{\mu}}(x,\mu_i(x))$ is the expected number of time steps until the next decision epoch of subtask M_i after taking action $\mu_i(x)$ in state x and following hierarchical policy $\boldsymbol{\mu}$ afterward, and $\omega \nearrow i$ is the content of the Task-Stack after popping subtask M_i off. Notice that $\hat{H}$ does not contain the average-adjusted rewards outside the current subtask and should

be distinguished from the hierarchical average-adjusted value function H which includes the sum of average-adjusted rewards outside the current subtask.

Since $r_i^{\boldsymbol{\mu}}(x, \mu_i(x))$ is the expected reward between two decision epochs of subtask M_i, given that the system occupies state x at the first decision epoch, and the agent chooses action $\mu_i(x)$, we have

$$r_i^{\boldsymbol{\mu}}(x, \mu_i(x)) = \hat{V}^{\boldsymbol{\mu}}(\mu_i(x), (\mu_i(x) \searrow \omega, s)) = \hat{H}^{\boldsymbol{\mu}}(\mu_i(x), (\mu_i(x) \searrow \omega, s)) + g^{\boldsymbol{\mu}} y_i^{\boldsymbol{\mu}}(x, \mu_i(x))$$

where $\mu_i(x) \searrow \omega$ is the content of the Task-Stack after pushing subtask $\mu_i(x)$ onto it. By replacing $r_i^{\boldsymbol{\mu}}(x, \mu_i(x))$ from the above expression, Equation 4.4 can be written as

$$H^{\boldsymbol{\mu}}(i, x) = \hat{H}^{\boldsymbol{\mu}}(\mu_i(x), (\mu_i(x) \searrow \omega, s)) + \\ \sum_{N_1, s_1 \in S_i} P_i^{\boldsymbol{\mu}}(x_1, N_1 | x, \mu_i(x)) \left[\hat{H}^{\boldsymbol{\mu}}(i, x_1) + \sum_{N_2, s_2 \in S_i} F_i^{\boldsymbol{\mu}}(x_2, N_2 | x_1) H^{\boldsymbol{\mu}}(Parent(i), (\omega \nearrow i, s_2)) \right] \tag{4.5}$$

We can restate Equation 4.5 for hierarchical average-adjusted action-value function as

$$L^{\boldsymbol{\mu}}(i, x, a) = \hat{H}^{\boldsymbol{\mu}}(a, (a \searrow \omega, s)) + \sum_{N_1, s_1 \in S_i} P_i^{\boldsymbol{\mu}}(x_1, N_1 | x, a) \\ \left[\hat{H}^{\boldsymbol{\mu}}(i, x_1) + \sum_{N_2, s_2 \in S_i} F_i^{\boldsymbol{\mu}}(x_2, N_2 | x_1) L^{\boldsymbol{\mu}}(Parent(i), (\omega \nearrow i, s_2), \mu_{parent(i)}(\omega \nearrow i, s_2)) \right] \tag{4.6}$$

From Equation 4.6, we can re-express the hierarchical average-adjusted action-value function L recursively as

$$L^{\boldsymbol{\mu}}(i, x, a) = \hat{H}^{\boldsymbol{\mu}}(a, (a \searrow \omega, s)) + C^{\boldsymbol{\mu}}(i, x, a) + CE^{\boldsymbol{\mu}}(i, x, a) \tag{4.7}$$

where

$$C^{\boldsymbol{\mu}}(i,x,a) = \sum_{N_1, s_1 \in S_i} P_i^{\boldsymbol{\mu}}(x_1, N_1|x,a)\hat{H}^{\boldsymbol{\mu}}(i,x_1) \tag{4.8}$$

and

$$\begin{aligned} CE^{\boldsymbol{\mu}}(i,x,a) = &\sum_{N_1, s_1 \in S_i} P_i^{\boldsymbol{\mu}}(x_1, N_1|x,a) \\ &\left[\sum_{N_2, s_2 \in S_i} F_i^{\boldsymbol{\mu}}(x_2, N_2|x_1) L^{\boldsymbol{\mu}}(Parent(i), (\omega \nearrow i, s_2), \mu_{parent(i)}(\omega \nearrow i, s_2))\right] \end{aligned} \tag{4.9}$$

The term $C^{\boldsymbol{\mu}}(i,x,a)$ is the expected average-adjusted reward of completing subtask M_i after executing action a in state $x = (\omega, s)$. We call this term **completion function** after Dietterich (2000). The term $CE^{\boldsymbol{\mu}}(i,x,a)$ is the expected average-adjusted reward received after subtask M_i terminates. We call this term **external completion function** after Andre and Russell (2002).

We can re-express the definition of $\hat{H}$ as

$$\hat{H}^{\boldsymbol{\mu}}(i,x) = \begin{cases} \hat{L}^{\boldsymbol{\mu}}(i,x,\mu_i(x)) & \text{if } M_i \text{ is a non-primitive subtask,} \\ r(s,i) - g^{\boldsymbol{\mu}} & \text{if } M_i \text{ is a primitive action.} \end{cases} \tag{4.10}$$

where $\hat{L}^{\boldsymbol{\mu}}$ is the projected average-adjusted action-value function and can be written as

$$\hat{L}^{\boldsymbol{\mu}}(i,x,a) = \hat{H}^{\boldsymbol{\mu}}(a,(a \searrow \omega, s)) + C^{\boldsymbol{\mu}}(i,x,a) \tag{4.11}$$

Equations 4.7 to 4.11 are the decomposition equations under a hierarchical policy $\boldsymbol{\mu}$. These equations recursively decompose the hierarchical average-adjusted value function for the *root*, $H^{\boldsymbol{\mu}}(0,x)$, into the projected average-adjusted value functions $\hat{H}^{\boldsymbol{\mu}}$ for

the individual subtasks $M_1, \ldots, M_{m-1}$ in the hierarchy,[3] the individual completion functions $C^{\boldsymbol{\mu}}(i, x, a)$ for $i = 1, \ldots, m-1$, and the individual external completion functions $CE^{\boldsymbol{\mu}}(i, x, a)$ for $i = 1, \ldots, m-1$. The fundamental quantities that must be stored to represent the hierarchical average-adjusted value function decomposition are the C and the CE values for all non-primitive subtasks, the $\hat{H}$ values for all primitive actions, and the *global gain* $g^{\boldsymbol{\mu}}$. The decomposition equations can be used to obtain update equations for $\hat{H}$, C, and CE in this hierarchically optimal average reward model. Pseudo-code for the discrete-time *hierarchically optimal average reward RL* (HAR) algorithm is shown in Algorithm 1. In this algorithm, primitive subtasks update their projected average-adjusted value functions $\hat{H}$ (Line 5), while non-primitive subtasks update both their completion functions C (Line 17), and external completion functions CE (Lines 19 and 21). We store only one global gain g and update it after each non-random primitive action (Line 7). In the update formula on Line 17, the projected average-adjusted value function $\hat{H}(a, (a \searrow \omega, s))$ is the reward of executing action a in state (ω, s) under subtask M_i and is recursively calculated by subtask M_a and its descendants using Equations 4.10 and 4.11. Notice that the hierarchical average-adjusted action-value function L on Lines 15 and 19 is recursively evaluated using Equation 4.7.

This algorithm can be easily extended to continuous-time by changing the update formulas for $\hat{H}$ and g on Lines 5 and 7 as

$$\hat{H}_{t+1}(i, x) \leftarrow [1 - \alpha_t(i)]\hat{H}_t(i, x) + \alpha_t(i)\left[k(s, i) + r(s, i)\tau(s, i) - g_t\tau(s, i)\right]$$

$$g_{t+1} = \frac{r_{t+1}}{t_{t+1}} = \frac{r_t + k(s, i) + r(s, i)\tau(s, i)}{t_t + \tau(s, i)}$$

[3] m is the total number of subtasks in the hierarchy.

Algorithm 1 Discrete-time hierarchically optimal average reward RL (HAR) algorithm.

Function HAR(Task M_i, State $x = (\omega, s)$)
let *Seq* ={}be the sequence of *states visited* while executing subtask M_i
if M_i is a primitive action **then**
    execute action i in state x, observe state $x' = (\omega, s')$ and reward $r(s, i)$
    $\hat{H}_{t+1}(i, x) \leftarrow [1 - \alpha_t(i)]\hat{H}_t(i, x) + \alpha_t(i)[r(s, i) - g_t]$
    if M_i and all its ancestors are non-random actions **then**
        update the global gain $g_{t+1} = \frac{r_{t+1}}{n_{t+1}} = \frac{r_t + r(s,i)}{n_t + 1}$
    end if
    push *state* $x_1 = (\omega \nearrow i, s)$ into the beginning of *Seq*
else /* M_i is a non-primitive subtask */
    while M_i has not terminated **do**
        choose action (subtask) a according to the current exploration policy $\mu_i(x)$
        let *ChildSeq*=HAR(M_a, $(a \searrow \omega, s)$), where *ChildSeq* is the sequence of states visited while executing subtask M_a
        observe result state $x' = (\omega, s')$
        let $a^* = \arg\max_{a' \in A_i(s')} L_t(i, x', a')$
        for each $x = (\omega, s)$ in *ChildSeq* from the beginning **do**
            $C_{t+1}(i, x, a) \leftarrow [1 - \alpha_t(i)]C_t(i, x, a) + \alpha_t(i) \left[\hat{H}_t(a^*, (a^* \searrow \omega, s')) + C_t(i, x', a^*)\right]$
            if $s' \in T_i$ (s' belongs to T_i the set of terminal states of subtask M_i) **then**
                $CE_{t+1}(i, x, a) \leftarrow [1 - \alpha_t(i)]CE_t(i, x, a) + \alpha_t(i)L_t(Parent(i), (\omega \nearrow i, s'), a^*)$
            else /* s' is not a terminal state of subtask M_i */
                $CE_{t+1}(i, x, a) \leftarrow [1 - \alpha_t(i)]CE_t(i, x, a) + \alpha_t(i)CE_t(i, x', a^*)$
            end if
            replace state $x = (\omega, s)$ with $(\omega \nearrow i, s)$ in the *ChildSeq*
        end for
        append *ChildSeq* onto the front of *Seq*
        $x = x'$
    end while
end if
return *Seq*
end HAR

where $\tau(s, i)$ is the time elapsing between state s and the next state, $k(s, i)$ is the fixed reward of taking action i in state s, and $r(s, i)$ is the reward rate for the time between state s and the next state.

4.2 Recursively Optimal Average Reward RL

In the previous section, we introduced discrete-time and continuous-time hierarchically optimal average reward RL (HAR) algorithms. In HAR algorithm, we define only a *global gain* for the entire hierarchy to guarantee hierarchical optimality for the overall task. The HAR algorithm finds a hierarchical policy that has the highest *global gain* among all policies consistent with the given hierarchy. However, there may exist subtasks where their policies must be locally suboptimal so that the overall policy becomes optimal. Recursive optimality is a kind of local optimality in which the policy at each node is optimal given the policies of its children (See Section 3.3). Thus, the goal at *root* is to maximize its gain given the policies for its descendants. The reason to seek recursive optimality rather than hierarchical optimality is that recursive optimality makes it possible to solve each subtask without reference to the context in which it is executed, and therefore the learned subtask can be reused by other hierarchies. This leaves open the question of what local optimality criterion should be used for each subtask in a recursively optimal average reward RL setting.

One approach pursued by Seri and Tadepalli (2002) is to optimize subtasks using their expected total average-adjusted reward with respect to *global gain*. Seri and Tadepalli introduced a model-based algorithm called *Hierarchical H-Learning* (HH-Learning). For every subtask, this algorithm learns the action model and maximizes the expected total average-adjusted reward with respect to *global gain* at each state. In this method, the projected average-adjusted value functions with respect to *global gain* satisfy the following equations:

$$\hat{H}^{\boldsymbol{\mu}}(i,s) = \begin{cases} r(s,i) - g^{\boldsymbol{\mu}} & \text{if } M_i \text{ is a primitive action,} \\ 0 & \text{if } s \text{ is a terminal state of subtask } M_i, \\ \max_{a \in A_i(s)}[\hat{H}^{\boldsymbol{\mu}}(a,s) + \sum_{N,s' \in S_i} P_i^{\boldsymbol{\mu}}(s',N|s,a)\hat{H}^{\boldsymbol{\mu}}(i,s')] & \text{otherwise.} \end{cases} \tag{4.12}$$

The first term of the last part of Equation 4.12, $\hat{H}^{\boldsymbol{\mu}}(a,s)$, denotes the expected total average-adjusted reward during the execution of subtask M_a (the projected average adjusted value function of subtask M_a), and the second term denotes the expected total average-adjusted reward from then on until the completion of subtask M_i (the completion function of subtask M_i after execution of subtask M_a). Since the expected average-adjusted reward after execution of subtask M_i is not a component of the average-adjusted value function of subtask M_i, this approach does not necessarily allow for hierarchical optimality, as we will show in the experiments of Section 4.3. Moreover, the policy learned for each subtask using this approach is not context free, since each subtask maximizes its average-adjusted reward with respect to *global gain*. However, Seri and Tadepalli (2002) showed that this method finds the hierarchically optimal average reward policy when the *result distribution invariance* (RDI) condition holds.

Definition 4.1 (Result Distribution Invariance (RDI) Condition): For all subtasks M_i and states s in the hierarchy, the distribution of states reached after the execution of any subtask M_a (M_a is one of M_i's children) is independent of the policy of subtask M_a, μ_a, and the policies of M_a's descendants, i.e., $P_i^{\boldsymbol{\mu}}(s'|s,a) = P_i(s'|s,a)$. □

In other words, states reached after the execution of a subtask cannot be changed by altering the policies of the subtask and its descendants. Note that the RDI condition does not hold for every problem, and therefore the HH-Learning algorithm is neither hierarchically nor recursively optimal in general.

Another approach is to formulate subtasks as continuing average reward problems, where the goal at each subtask is to maximize its gain given the policies of its children (Ghavamzadeh and Mahadevan, 2001). We first describe this approach in detail in Sections 4.2.1 and 4.2.2. In Section 4.2.3, we use this method to find recursively optimal average reward policies, and present discrete-time and continuous-time *recursively optimal average reward RL* (RAR) algorithms. Finally in Section 4.2.4, we investigate the conditions under which the policy learned by the RAR algorithm at each subtask is independent of the context in which it is executed and therefore can be reused by other hierarchies.

4.2.1 Root Task Formulation

In our approach, we consider those problems for which Assumption 4.1 (*Continuing Root Task*) and the following assumption hold.

Assumption 4.3 (Root Task Recurrence): There exists a state $s_0^* \in S_0$ such that, for every hierarchical policy $\boldsymbol{\mu}$ and for every state $s \in S_0$, we have[4]

$$\sum_{N=1}^{|S_0|} F_0^{\boldsymbol{\mu}}(s_0^*, N|s) > 0$$

where $F_0^{\boldsymbol{\mu}}$ is the multi-step abstract transition probability function of *root* under the hierarchical policy $\boldsymbol{\mu}$ described in Section 3.2, and $|S_0|$ is the number of states in the state space of *root*. □

Assumption 4.3 is equivalent to assuming that the underlying Markov chain at *root* for every hierarchical policy $\boldsymbol{\mu}$ has a single recurrent class, and state s_0^* is a recurrent state. The recurrent state s_0^* can be a terminal state of any of *root*'s children. If Assumptions 4.1

[4]Notice that the *root* task is represented as subtask M_0 in the HRL framework described in Chapter 3. So we use index 0 to represent every component of the *root* task.

and 4.3 hold, the gain at the *root* task under the hierarchical policy $\boldsymbol{\mu}$, $g_0^{\boldsymbol{\mu}}$, is well defined for every hierarchical policy $\boldsymbol{\mu}$ and does not depend on the initial state. When the state space at *root* is finite or countable, the average reward or gain at *root* can be written as

$$g_0^{\boldsymbol{\mu}} = \frac{\bar{\boldsymbol{m}}_0^{\boldsymbol{\mu}} r_0^{\boldsymbol{\mu}}(s, \mu_0(s))}{\bar{\boldsymbol{m}}_0^{\boldsymbol{\mu}} y_0^{\boldsymbol{\mu}}(s, \mu_0(s))}$$

where $r_0^{\boldsymbol{\mu}}(s, \mu_0(s))$ and $y_0^{\boldsymbol{\mu}}(s, \mu_0(s))$ denote the expected total reward and the expected number of time steps between two decision epochs at *root*, given that the system occupies state s at the first decision epoch and the agent chooses its actions according to the hierarchical policy $\boldsymbol{\mu}$. The terms $\boldsymbol{m}_0^{\boldsymbol{\mu}}$ and $\bar{\boldsymbol{m}}_0^{\boldsymbol{\mu}} = \lim_{N\to\infty} \frac{1}{N} \sum_{t=0}^{N-1} (\boldsymbol{m}_0^{\boldsymbol{\mu}})^t$ are the transition probability matrix and the *limiting matrix* of the embedded Markov chain at *root* for the hierarchical policy $\boldsymbol{\mu}$ respectively. The transition probability $\boldsymbol{m}_0^{\boldsymbol{\mu}}$ is obtained by marginalizing the multi-step abstract transition probability $F_0^{\boldsymbol{\mu}}$. The term $\boldsymbol{m}_0^{\boldsymbol{\mu}}(s'|s, \mu_0(s))$ denotes the probability that the SMDP at *root* occupies state s' at the next decision epoch, given that the agent chooses action $\mu_0(s)$ in state s at the current decision epoch and follows the hierarchical policy $\boldsymbol{\mu}$.

4.2.2 Subtask Formulation

In Section 4.2.1, we described the average reward formulation of the *root* task of a hierarchical decomposition. In this section, we illustrate how we formulate all other subtasks in a hierarchy as average reward problems. From now on in this chapter, we use subtask to refer to non-primitive subtasks in a hierarchy except *root*.

In the HRL methods, we typically assume that every time a subtask M_i is executed, it starts at one of its initial states ($\in \mathcal{I}_i$) and terminates at one of its terminal states ($\in T_i$) after a finite number of steps. Therefore, we can make the following assumption for every subtask M_i in a hierarchy. Under this assumption, each subtask can be considered an episodic problem and each instantiation of a subtask can be considered an episode.

Assumption 4.4 (Subtask Termination): There exists a dummy state $s_i^* \in S_i$ such that, for every action $a \in A_i$ and every terminal state s_{T_i}, we have

$$r_i(s_{T_i}, a) = 0 \quad \text{and} \quad P_i(s_i^*, 1|s_{T_i}, a) = 1$$

and for all hierarchical stationary policies $\boldsymbol{\mu}$ and non-terminal states $s \in S_i$, we have

$$F_i^{\boldsymbol{\mu}}(s_i^*, 1|s) = 0$$

and finally for all states $s \in S_i$, we have

$$F_i^{\boldsymbol{\mu}}(s_i^*, N|s) > 0$$

where $F_i^{\boldsymbol{\mu}}$ is the multi-step abstract transition probability function of subtask M_i under the hierarchical policy $\boldsymbol{\mu}$ described in Section 3.2, and $N = |S_i|$ is the number of states in the state space of subtask M_i. $\square$

Although subtasks are episodic problems, when the overall task (*root* of the hierarchy) is continuing as we assumed in this chapter (Assumption 4.1), they are executed infinite number of times, and therefore can be modeled as continuing problems using the model described in Figure 4.1. In this model, each subtask M_i terminates at one of its terminal states $s_{T_i} \in T_i$. All terminal states transit with probability 1 and reward 0 to a dummy state s_i^*. This is a dummy transition and does not add a time-step to the cycle of subtask M_i and therefore is not taken into consideration when the average reward of subtask M_i is calculated. Finally, the dummy state s_i^* transits with reward zero to one of the initial states ($\in \mathcal{I}_i$) of subtask M_i upon the next instantiation of M_i. It is important for the validity of this model to fix the value of dummy states to zero.

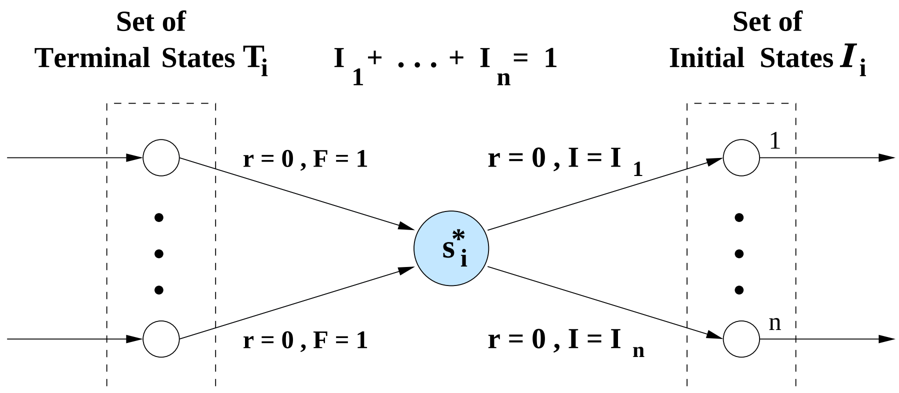


Figure 4.1. This figure shows how each subtask in a hierarchical decomposition of a continuing problem can be modeled as a continuing task.

Under this model, for every hierarchical policy $\boldsymbol{\mu}$, each subtask M_i in the hierarchy can be modeled using a new MDP with abstract transition probabilities and rewards

$$F_{I_i}^{\boldsymbol{\mu}}(s', 1|s) = \begin{cases} F_i^{\boldsymbol{\mu}}(s', 1|s) & s \neq s_i^*, \\ I_i(s') & s = s_i^*. \end{cases} \tag{4.13}$$

$$r_{I_i}^{\boldsymbol{\mu}}(s, a) = r_i^{\boldsymbol{\mu}}(s, a)$$

where $I_i(s)$ is the probability that subtask M_i starts at state s.

Let $\mathcal{F}_{I_i}^{\boldsymbol{\mu}}$ be the set of all abstract transition probability functions $F_{I_i}^{\boldsymbol{\mu}}$. We have the following result for subtask M_i.

Lemma 4.1: Let Assumption 4.4 (*Subtask Termination*) hold. Then for every $F_{I_i}^{\boldsymbol{\mu}} \in \mathcal{F}_{I_i}^{\boldsymbol{\mu}}$ and every state $s \in S_i$, we have $\sum_{N=1}^{|S_i|} F_{I_i}^{\boldsymbol{\mu}}(s_i^*, N|s) > 0$.[5] □

Lemma 4.1 is equivalent to assuming that for every subtask M_i in the hierarchy, the un-

[5] This lemma is a restatement of the Lemma 5 on page 34 of Peter Marbach's thesis (Marbach, 1998).

derlying Markov chain for every hierarchical policy $\boldsymbol{\mu}$ has a single recurrent class and state s_i^* is its recurrent state. Under this model, the gain of subtask M_i under the hierarchical policy $\boldsymbol{\mu}$, $g_i^{\boldsymbol{\mu}}$, is well defined for every hierarchical policy $\boldsymbol{\mu}$ and does not depend on the initial state. When the state space of subtask M_i is finite or countable, the gain of subtask M_i can be written as

$$g_i^{\boldsymbol{\mu}} = \frac{\bar{\boldsymbol{m}}_{I_i}^{\boldsymbol{\mu}} r_{I_i}^{\boldsymbol{\mu}}(s, \mu_i(s))}{\bar{\boldsymbol{m}}_{I_i}^{\boldsymbol{\mu}} y_{I_i}^{\boldsymbol{\mu}}(s, \mu_i(s))}$$

where $r_{I_i}^{\boldsymbol{\mu}}(s, \mu_i(s))$ and $y_{I_i}^{\boldsymbol{\mu}}(s, \mu_i(s))$ are equal to $r_i^{\boldsymbol{\mu}}(s, \mu_i(s))$ and $y_i^{\boldsymbol{\mu}}(s, \mu_i(s))$, and denote the expected total reward and the expected number of time steps between two decision epochs of subtask M_i, given that the system occupies state s at the first decision epoch and the agent chooses its actions according to the hierarchical policy $\boldsymbol{\mu}$. The terms $\boldsymbol{m}_{I_i}^{\boldsymbol{\mu}}$ and $\bar{\boldsymbol{m}}_{I_i}^{\boldsymbol{\mu}} = \lim_{N\to\infty} \frac{1}{N} \sum_{t=0}^{N-1} (\boldsymbol{m}_{I_i}^{\boldsymbol{\mu}})^t$ are the transition probability matrix and the *limiting matrix* of the Markov chain[6] at subtask M_i for the hierarchical policy $\boldsymbol{\mu}$ respectively. The transition probability $\boldsymbol{m}_{I_i}^{\boldsymbol{\mu}}$ is obtained by marginalizing the multi-step abstract transition probability $F_{I_i}^{\boldsymbol{\mu}}$.

4.2.3 Recursively Optimal Average Reward RL Algorithm

In this section, we present discrete-time and continuous-time recursively optimal average reward RL (RAR) algorithms using the formulation described in Sections 4.2.1 and 4.2.2. We consider problems for which Assumptions 4.1, 4.3, and 4.4 (*Continuing Root Task*, *Root Task Recurrence*, and *Subtask Termination*) hold, *root* is modeled as an average reward problem as described in Section 4.2.1, and every other non-primitive subtask in the hierarchy is modeled as an average reward problem using the model described in Section 4.2.2. Under these assumptions, the average reward for every non-primitive subtask in the hierarchy including *root* is well defined for every hierarchical policy and does not vary with

[6]This Markov chain corresponds to the MDP at subtask M_i defined by Equation 4.13, not the original MDP at subtask M_i defined by $F_i^{\boldsymbol{\mu}}$ and $r_i^{\boldsymbol{\mu}}$.

initial state. Since we are interested in finding a recursively optimal average reward policy, we do not need to include the contents of the Task-Stack as a part of the state space of the problem. We also replace projected value and action-value functions in the hierarchical model of Chapter 3 with projected average-adjusted value and projected average-adjusted action-value functions described in Sections 2.2.3 and 2.3.2.

We show how the overall projected average-adjusted value function $\hat{H}^{\boldsymbol{\mu}}(0,s)$ is decomposed into a collection of projected average-adjusted value functions of individual subtasks $\hat{H}^{\boldsymbol{\mu}}(i,s)$ for $i=1,\ldots,m-1$, in the RAR algorithm. The projected average-adjusted value function of hierarchical policy $\boldsymbol{\mu}$ on subtask M_i is the average-adjusted (with respect to the local gain $g_i^{\boldsymbol{\mu}}$) sum of rewards earned by following policy μ_i and the policies of all descendants of subtask M_i starting in state s until subtask M_i terminates. Now let us suppose that the first action chosen by μ_i is invoked and executed for a number of primitive steps N and terminates in state s' according to multi-step transition probability $P_i^{\boldsymbol{\mu}}(s',N|s,\mu_i(s))$. We can write the projected average-adjusted value function in the form of a Bellman equation as

$$\hat{H}^{\boldsymbol{\mu}}(i,s) = r_i^{\boldsymbol{\mu}}(s,\mu_i(s)) - g_i^{\boldsymbol{\mu}} y_i^{\boldsymbol{\mu}}(s,\mu_i(s)) + \sum_{N,s'\in S_i} P_i^{\boldsymbol{\mu}}(s',N|s,\mu_i(s))\hat{H}^{\boldsymbol{\mu}}(i,s') \quad (4.14)$$

Since the term $r_i^{\boldsymbol{\mu}}(s,\mu_i(s))$ is the expected total reward between two decision epochs of subtask M_i, given that the system occupies state s at the first decision epoch, the agent chooses action $\mu_i(s)$, and the number of time steps until the next decision epoch is defined by $y_i^{\boldsymbol{\mu}}(s,\mu_i(s))$, we have

$$r_i^{\boldsymbol{\mu}}(s,\mu_i(s)) = \begin{cases} \hat{V}^{\boldsymbol{\mu}}(\mu_i(s),s) = \hat{H}^{\boldsymbol{\mu}}(\mu_i(s),s) + g^{\boldsymbol{\mu}}_{\mu_i(s)} y_i^{\boldsymbol{\mu}}(s,\mu_i(s)) & \\ & \text{if } M_i \text{ is a non-primitive subtask,} \\ \hat{V}^{\boldsymbol{\mu}}(\mu_i(s),s) & \\ & \text{if } M_i \text{ is a primitive action.} \end{cases}$$

By replacing $r_i^{\boldsymbol{\mu}}(s, \mu_i(s))$ from the above expression, Equation 4.14 can be written as

$$\hat{H}^{\boldsymbol{\mu}}(i,s) = \begin{cases} \hat{H}^{\boldsymbol{\mu}}(\mu_i(s), s) - (g_i^{\boldsymbol{\mu}} - g_{\mu_i(s)}^{\boldsymbol{\mu}}) y_i^{\boldsymbol{\mu}}(s, \mu_i(s)) + \sum_{N, s' \in S_i} P_i^{\boldsymbol{\mu}}(s', N | s, \mu_i(s)) \hat{H}^{\boldsymbol{\mu}}(i, s') \\ \qquad \text{if } M_i \text{ is a non-primitive subtask,} \\ \hat{V}^{\boldsymbol{\mu}}(\mu_i(s), s) - g_i^{\boldsymbol{\mu}} y_i^{\boldsymbol{\mu}}(s, \mu_i(s)) + \sum_{N, s' \in S_i} P_i^{\boldsymbol{\mu}}(s', N | s, \mu_i(s)) \hat{H}^{\boldsymbol{\mu}}(i, s') \\ \qquad \text{if } M_i \text{ is a primitive action.} \end{cases} \tag{4.15}$$

We can re-state Equations 4.15 for projected action-value function as follows:

$$\hat{L}^{\boldsymbol{\mu}}(i,s,a) = \begin{cases} \hat{H}^{\boldsymbol{\mu}}(a, s) - (g_i^{\boldsymbol{\mu}} - g_a^{\boldsymbol{\mu}}) y_i^{\boldsymbol{\mu}}(s, a) + \sum_{N, s' \in S_i} P_i^{\boldsymbol{\mu}}(s', N | s, a) \hat{L}^{\boldsymbol{\mu}}(i, s', \mu_i(s')) \\ \qquad \text{if } M_i \text{ is a non-primitive subtask,} \\ \hat{V}^{\boldsymbol{\mu}}(a, s) - g_i^{\boldsymbol{\mu}} y_i^{\boldsymbol{\mu}}(s, a) + \sum_{N, s' \in S_i} P_i^{\boldsymbol{\mu}}(s', N | s, a) \hat{L}^{\boldsymbol{\mu}}(i, s', \mu_i(s')) \\ \qquad \text{if } M_i \text{ is a primitive action.} \end{cases} \tag{4.16}$$

In the above equation, when M_i is a non-primitive subtask, the term

$$-(g_i^{\boldsymbol{\mu}} - g_a^{\boldsymbol{\mu}}) y_i^{\boldsymbol{\mu}}(s, a) + \sum_{N, s' \in S_i} P_i^{\boldsymbol{\mu}}(s', N | s, a) \hat{L}^{\boldsymbol{\mu}}(i, s', \mu_i(s'))$$

and when M_i is a primitive action, the term

$$-g_i^{\boldsymbol{\mu}} y_i^{\boldsymbol{\mu}}(s, a) + \sum_{N, s' \in S_i} P_i^{\boldsymbol{\mu}}(s', N | s, a) \hat{L}^{\boldsymbol{\mu}}(i, s', \mu_i(s'))$$

denote the average-adjusted reward of completing subtask M_i after executing action a in state s. We call this term **completion function** after Dietterich (2000), and denote it by

$C^{\boldsymbol{\mu}}(i, s, a)$. With this definition, we can express the average-adjusted action-value function $\hat{L}^{\boldsymbol{\mu}}$ recursively as

$$\hat{L}^{\boldsymbol{\mu}}(i, s, a) = \begin{cases} \hat{H}^{\boldsymbol{\mu}}(a, s) + C^{\boldsymbol{\mu}}(i, s, a) & \text{if } M_i \text{ is a non-primitive subtask,} \\ \hat{V}^{\boldsymbol{\mu}}(a, s) + C^{\boldsymbol{\mu}}(i, s, a) & \text{if } M_i \text{ is a primitive action.} \end{cases} \tag{4.17}$$

where

$$C^{\boldsymbol{\mu}}(i, s, a) = \begin{cases} -(g_i^{\boldsymbol{\mu}} - g_a^{\boldsymbol{\mu}}) y_i^{\boldsymbol{\mu}}(s, a) + \sum_{N, s' \in S_i} P_i^{\boldsymbol{\mu}}(s', N|s, a) \hat{L}^{\boldsymbol{\mu}}(i, s', \mu_i(s')) \\ \qquad\qquad\qquad\qquad \text{if } M_i \text{ is a non-primitive subtask,} \\ -g_i^{\boldsymbol{\mu}} y_i^{\boldsymbol{\mu}}(s, a) + \sum_{N, s' \in S_i} P_i^{\boldsymbol{\mu}}(s', N|s, a) \hat{L}^{\boldsymbol{\mu}}(i, s', \mu_i(s')) \\ \qquad\qquad\qquad\qquad \text{if } M_i \text{ is a primitive action.} \end{cases} \tag{4.18}$$

and

$$\hat{H}^{\boldsymbol{\mu}}(i, s) = \hat{L}^{\boldsymbol{\mu}}(i, s, \mu_i(s)) \tag{4.19}$$

when M_i is a non-primitive subtask.

Equations 4.15 to 4.19 are the decomposition equations for projected average-adjusted value and projected average-adjusted action-value functions. They can be used to obtain update formulas for $\hat{H}$ and C in this recursively optimal average reward model. Pseudo-code for the discrete-time *recursively optimal average reward RL* (RAR) algorithm is shown in Algorithm 2. In this algorithm, a gain is defined for every non-primitive subtask in the hierarchy and this gain is updated every time a subtask is non-randomly chosen. Primitive subtasks store their projected value functions, and update them using the equation on Line 5. Non-primitive subtasks store their completion functions and gains, and update them using equations on Lines 17, 19, and 23. The projected average-adjusted action-value function $\hat{L}$ on Lines 12, 17, and 19 are recursively calculated using Equations 4.17 to 4.19.

Algorithm 2 Discrete-time recursively optimal average reward RL (RAR) algorithm.

Function RAR(Task M_i, State s)
let *Seq* ={}be the sequence of (*state visited*, *reward*) while executing subtask M_i
if M_i is a primitive action **then**
    execute action i in state s, observe state s' and reward $r(s,i)$
    $\hat{V}_{t+1}(i,s) \leftarrow [1-\alpha_t(i)]\hat{V}_t(i,s) + \alpha_t(i)r(s,i)$
    push (*state* s, *reward* $r(s,i)$) into the beginning of *Seq*
else /* M_i is a non-primitive subtask */
    while M_i has not terminated **do**
        choose action (subtask) a according to the current exploration policy $\mu_i(s)$
        let *ChildSeq*=RAR(M_a, s), where *ChildSeq* is the sequence of (*state visited*, *reward*) while executing subtask M_a
        observe result state s'
        let $a^* = \arg\max_{a' \in A_i(s')} \hat{L}_t(i,s',a')$
        let $N = 0$; $\rho = 0$;
        for each (s, r) in *ChildSeq* from the beginning **do**
            $N = N + 1$; $\rho = \rho + r$;
            if a is a primitive action **then**
                $C_{t+1}(i,s,a) \leftarrow [1-\alpha_t(i)]C_t(i,s,a) + \alpha_t(i)[\hat{L}_t(i,s',a^*) - g_t(i)N]$
            else /* M_a is a non-primitive subtask */
                $C_{t+1}(i,s,a) \leftarrow [1-\alpha_t(i)]C_t(i,s,a)+\alpha_t(i)\{\hat{L}_t(i,s',a^*)-[g_t(i)-g_t(a)]N\}$
            end if
        end for
        if M_a and all its ancestors are non-random actions **then**
            update gain of subtask M_i $\quad g_{t+1}(i) = \frac{r_{t+1}(i)}{n_{t+1}(i)} = \frac{r_t(i)+\rho}{n_t(i)+N}$
        end if
        append *ChildSeq* onto the front of *Seq*
        $s = s'$
    end while
end if
return *Seq*
end RAR

This algorithm can be easily extended to continuous-time. In the continuous-time version of the RAR algorithm, in addition to visited state and reward, we need to push the execution time of primitive actions τ into the *Seq*. Therefore $N = N + 1$ on Line 15 of the algorithm is replaced by $T = T + \tau$. We also need to modify the update formulas for $\hat{V}$, C, and g_i on Lines 5, 17, 19, and 23 as

$$\hat{V}_{t+1}(i,s) \leftarrow [1-\alpha_t(i)]\hat{H}_t(i,s) + \alpha_t(i)\left[k(s,i) + r(s,i)\tau(s,i)\right]$$

$$C_{t+1}(i,s,a) \leftarrow [1-\alpha_t(i)]C_t(i,s,a) + \alpha_t(i)[\hat{L}_t(i,s',a^*) - g_t(i)T]$$

$$C_{t+1}(i,s,a) \leftarrow [1-\alpha_t(i)]C_t(i,s,a) + \alpha_t(i)[\hat{L}_t(i,s',a^*) - (g_t(i) - g_t(a))T]$$

$$g_{t+1}(i) = \frac{r_{t+1}(i)}{t_{t+1}(i)} = \frac{r_t(i) + \rho}{t_t(i) + T}$$

where $\tau(s,i)$ is the time elapsing between state s and the next state, $k(s,i)$ is the fixed reward of taking action i in state s, and $r(s,i)$ is the reward rate for the time between state s and the next state.

4.2.4 Optimality of the RAR Algorithm

In this section, we investigate the optimality achieved by the RAR algorithm. In the RAR algorithm, since the expected average-adjusted reward after execution of subtask M_i is not a component of the average-adjusted value function of subtask M_i, the algorithm fails to find a hierarchically optimal average reward policy in general, as it has been discussed in (Seri and Tadepalli, 2002) and we will demonstrate it in the experiments of Section 4.3.

To achieve recursive optimality, the policy learned for each subtask must be context free, that is, each subtask maximizes its local gain given the policies of its descendants. In

RAR algorithm, although each subtask maximizes its local gain given the policies of its descendants, the policy learned for each subtask is not necessarily context free, and as a result the algorithm does not find a recursively optimal average reward policy in general. The reason for that is, the local gain g_i for each subtask M_i does not depend only on the policies of its descendants. The local gain g_i is the gain of the SMDP defined by Equation 4.13 and therefore depends on the initial state distribution $I_i(s)$. The initial state distribution $I_i(s)$ depends not only on the policies of M_i's descendants, but also on the policies of its parents, which makes the local gain g_i context dependent. However, the algorithm finds a recursively optimal average reward policy when the *initial distribution invariance* (IDI) condition holds. In this case, the policy learned by this method at each subtask is independent of the context in which it is executed and therefore can be reused by other hierarchies.

Definition 4.2 (Initial Distribution Invariance (IDI) Condition): The initial state distribution for each non-primitive subtask in the hierarchy is independent of the policies of its parents. □

In other words, the initial state distribution for each non-primitive subtask cannot be changed by altering the policies of its parents. One special case that satisfies the IDI condition is when each non-primitive subtask in the hierarchy has only one initiation state, $|\mathcal{I}_i| = 1$ for $i = 1, \ldots, m-1$, and M_i is a non-primitive subtask.

4.3 Experimental Results

The goal of this section is to demonstrate the efficacy of the algorithms proposed in Sections 4.1 and 4.2. We show the type of optimality that they converge to as well as their performance and speed comparing to other algorithms. We conduct two sets of experiments in this section. In Section 4.3.1, we apply five HRL algorithms to a simple discrete-time AGV scheduling problem. The advantage of using this simple domain is that it clearly

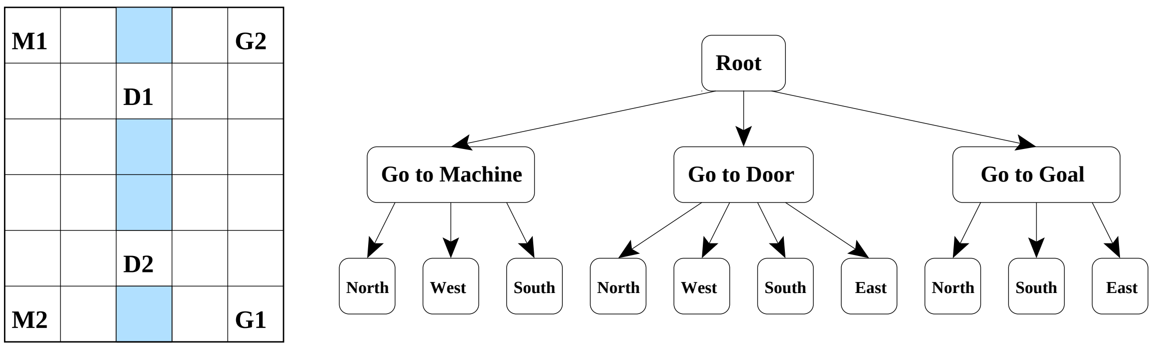
demonstrates the difference between the optimality criteria achieved by these algorithms. Then we turn to a more complex AGV scheduling task in Section 4.3.2. In this section, we model an AGV scheduling task as discrete time and continuous time problems and apply HAR and RAR algorithms as well as a flat average reward RL algorithm to both models.

4.3.1 A Small AGV Scheduling Problem

In this section, we apply the *discrete-time hierarchically optimal average reward RL* (HAR) algorithm described in Section 4.1, the *discrete-time recursively optimal average reward RL* (RAR) algorithm described in Section 4.2, and *HH-Learning*, the algorithm proposed by Seri and Tadepalli (2002), to a small AGV scheduling task. We also test MAXQ-Q, the recursively optimal discounted reward HRL algorithm proposed by Dietterich (2000), and a *hierarchically optimal discounted reward RL* algorithm (HDR) on this task. The HDR algorithm is an extension of the MAXQ-Q using the three-part value function decomposition proposed by Andre and Russell (2002) described in Chapter 3. These experimental results clearly demonstrate the difference between the optimality criteria achieved by these algorithms.

A small AGV domain is depicted in Figure 4.2. In this domain there are two machines $M1$ and $M2$ that produce parts to be delivered to corresponding destination stations $G1$ and $G2$. Since machines and destination stations are in two different rooms, the AGV has to pass one of the two doors $D1$ and $D2$ every time it goes from one room to another. Part 1 is more important than part 2, therefore the AGV gets a reward of 20 when part 1 is delivered to destination $G1$ and a reward of 1 when part 2 is delivered to destination $G2$. The AGV receives a reward of -1 for all other actions. This task is deterministic and the state variables are *AGV location* and *AGV status* (empty, carry part 1 or carry part 2), which is total of $26 \times 3 = 78$ states. In all experiments, we use the task graph shown in Figure 4.2 and set the discount factor to 0.99 for the discounted reward algorithms. We tried several discounting factors and 0.99 yielded the best performance. Using this task graph,

hierarchical and recursive optimal policies are different. Since delivering part 1 has more reward than part 2, the hierarchically optimal policy is one in which the AGV always serves machine $M1$. In the recursively optimal policy, the AGV switches from serving machine $M1$ to serving machine $M2$ and vice versa. In this policy, the AGV goes to machine $M1$, picks up a part of type 1, goes to goal $G1$ via door $D1$, drops the part there, then passes through door $D2$, goes to machine $M2$, picks up a part of type 2, goes to goal $G2$ via door $D2$ and then switches again to machine $M1$ and so on so forth.



M1: Machine 1 **M2:** Machine 2 **D1:** Door 1 **D2:** Door 2 **G1:** Goal 1 **G2:** Goal 2

Figure 4.2. A small AGV scheduling task and its associated task graph.

Among the algorithms we applied to this task, the hierarchically optimal average reward RL (HAR) and the hierarchically optimal discounted reward RL (HDR) algorithms find the hierarchically optimal policy, where the other algorithms only learn the recursively optimal policy. Figure 4.3 demonstrates the throughput of the system for the above algorithms. In this figure, the throughput of the system is the number of parts deposited at the destination stations weighted by their reward ($part1 \times 20 + part2 \times 1$) in $10,000$ time steps. Each experiment was conducted ten times and the results were averaged.

4.3.2 AGV Scheduling Problem (Discrete and Continuous Time Models)

In this section, we describe two sets of experiments on an AGV scheduling problem shown in Figure 4.4. $M1$ to $M3$ are workstations in this environment. Parts of type i have

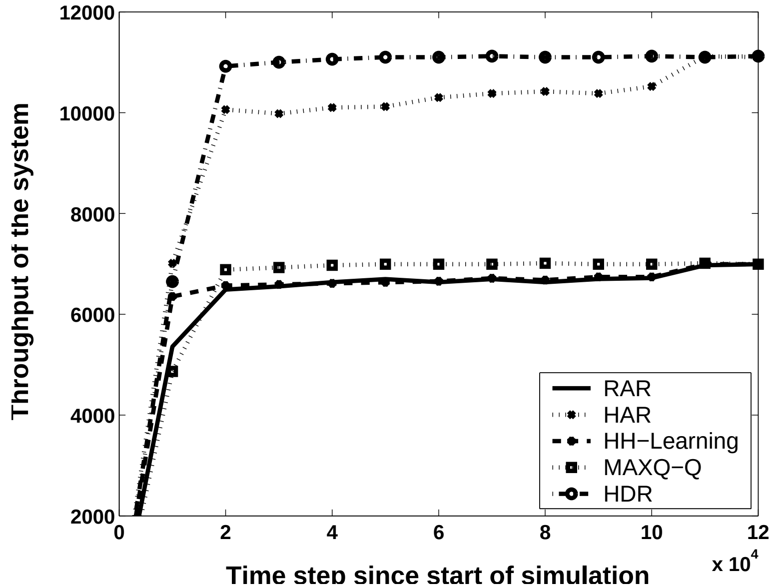


Figure 4.3. This plot shows that HDR and HAR algorithms (the two curves at the top) learn the hierarchically optimal policy while RAR, MAXQ-Q, and HH-Learning (the three curves at the bottom) only find the recursively optimal policy for the small AGV scheduling task.

to be carried to the drop-off station at workstation i (D_i), and the assembled parts brought back from pick-up stations of workstations (P_i's) to the warehouse. The AGV travel is unidirectional as the arrows show. We model this AGV scheduling task using both discrete-time and continuous-time models and demonstrate the performance and speed of three HRL algorithms: *hierarchically optimal average reward RL* (HAR), *recursively optimal average reward RL* (RAR), and *hierarchically optimal discounted reward RL* (HDR) as well as a *non-hierarchical average reward* algorithm in this problem. In both experiments, we use the task graph shown in Figure 4.5 for the AGV scheduling problem, and discount factors 0.9 and 0.95 for discounted reward algorithms. Using discount factor 0.95 yielded better performance in both experiments.

The state of the environment consists of the number of parts in the pick-up and drop-off stations of each machine, and whether the warehouse contains parts of each of the three types. In addition, agent keeps track of its own location and status as a part of its state

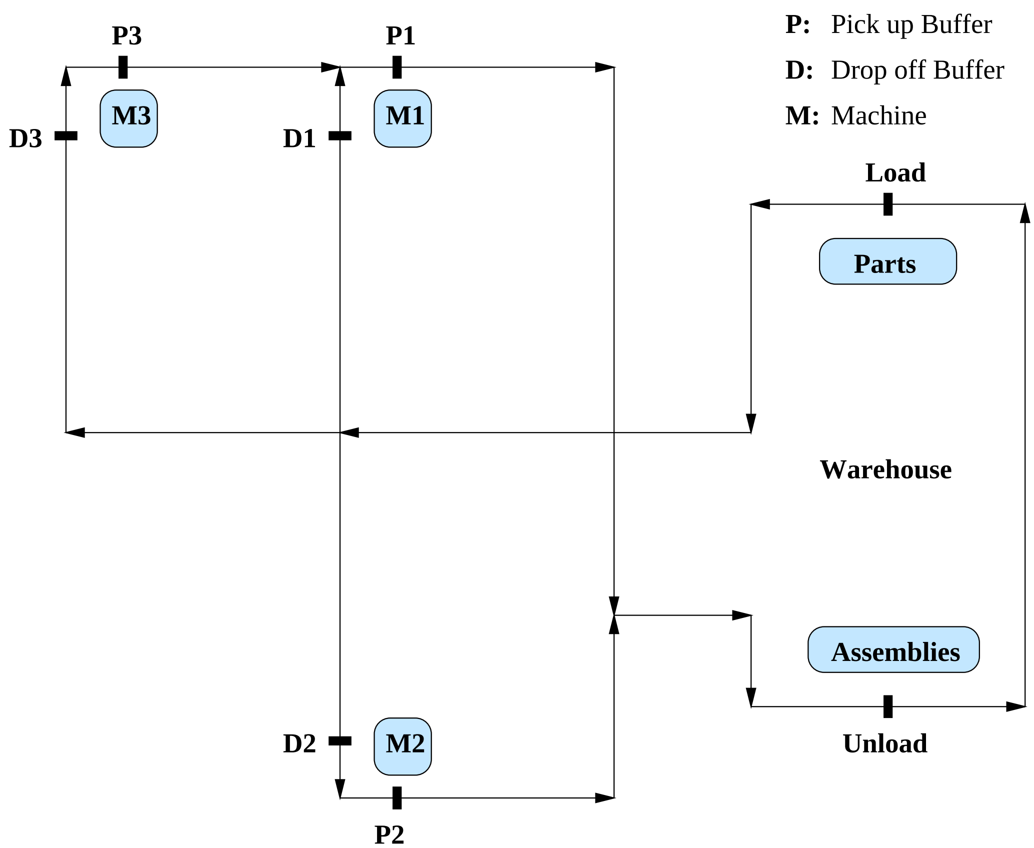


Figure 4.4. An AGV scheduling task. An AGV agent (not shown) carries raw materials and finished parts between machines and warehouse.

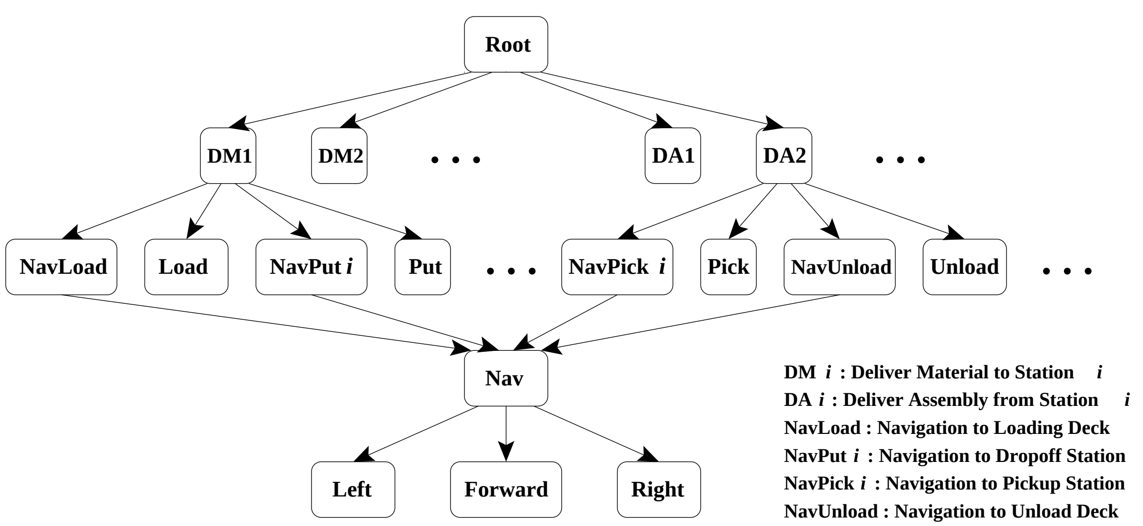


Figure 4.5. Task graph for the AGV scheduling task.

space. Thus in the flat case, state space consists of 33 locations, 6 buffers of size 2, 7 possible states of the AGV (carrying Part1, $\ldots$, carrying Assembly1, $\ldots$, empty), and 2 values for each part in the warehouse, i.e., $33 \times 3^6 \times 7 \times 2^3 = 1,347,192$ states. *State abstraction* helps in reducing the state space considerably. Only the relevant state variables are used while storing the value functions in each node of the task graph. For example, for the *Navigation* subtask, only the location state variable is relevant and this subtask can be learned with only 33 values. Hence for each of the high-level subtasks $DM1, \ldots, DM3$, the number of relevant states would be $33 \times 7 \times 3 \times 2 = 1,386$, and for each of the high-level subtasks $DA1, \ldots, DA3$, the number of relevant states would be $33 \times 7 \times 3 = 693$. This state abstraction gives us a compact way of representing the value functions and speeds up the algorithm.

The discrete-time experimental results were generated with the following model parameters. The inter-arrival time for parts at the warehouse is uniformly distributed with a mean of 12 time steps and variance of 2 time steps. The percentage of *Part1*, *Part2*, and *Part3* in the part arrival process are 40, 35, and 25 respectively. The time required for assembling the various parts are Poisson random variables with means 6, 10, and 12 time steps for *Part1*, *Part2*, and *Part3* respectively, and variance 2 time steps. Table 4.1 shows the parameters of the discrete-time model.

Parameter	**Distribution**	**Mean (steps)**	**Variance (steps)**
Assembly Time for Part1	Poisson	6	2
Assembly Time for Part2	Poisson	10	2
Assembly Time for Part3	Poisson	12	2
Inter-Arrival Time for Parts	Uniform	12	2

Table 4.1. Parameters of the Discrete-Time Model

The continuous-time experimental results were generated with the following model parameters. The time required for execution of each primitive action is a normal random variable with mean 10 seconds and variance 2 seconds. The inter-arrival time for parts at the warehouse is uniformly distributed with a mean of 100 seconds and variance of 20

seconds. The percentage of *Part1*, *Part2*, and *Part3* in the part arrival process are 40, 35, and 25 respectively. The time required for assembling the various parts are normal random variables with means 100, 120, and 180 seconds for *Part1*, *Part2*, and *Part3* respectively, and variance 20 seconds. Table 4.2 contains the parameters of the continuous-time model. In both cases, each experiment was conducted five times and the results were averaged.

Parameter	**Type of Distribution**	**Mean (sec)**	**Variance (sec)**
Execution Time for Primitive Actions	Normal	10	2
Assembly Time for Part1	Normal	100	20
Assembly Time for Part2	Normal	120	20
Assembly Time for Part3	Normal	180	20
Inter-Arrival Time for Parts	Uniform	100	20

Table 4.2. Parameters of the Continuous-Time Model

Figure 4.6 compares the discrete-time hierarchically optimal average reward RL (HAR) algorithm described in Section 4.1 with the discrete-time discounted reward hierarchically optimal (HDR) algorithm, and the discrete-time recursively optimal average reward RL (RAR) algorithm illustrated in Section 4.2. The graph shows the improved performance of the HAR algorithm. This figure also shows that the HAR algorithm converges faster to the same throughput as the non-hierarchical average reward algorithm. The non-hierarchical average reward algorithm used in this experiment is relative value iteration (RVI) Q-learning (Abounadi et al., 2001). The difference in convergence speed between flat and hierarchical algorithms becomes more significant as we increase the number of states.

Figure 4.7 compares the continuous-time hierarchically optimal average reward RL (HAR) algorithm described in Section 4.1 with the continuous-time hierarchically optimal discounted reward RL (HDR) algorithm, and the continuous-time recursively optimal average reward RL (RAR) algorithm illustrated in Section 4.2. The graph shows that the HAR algorithm converges to the same performance as the discounted reward HDR algorithm, and both have better performance than the RAR (recursively optimal average reward)

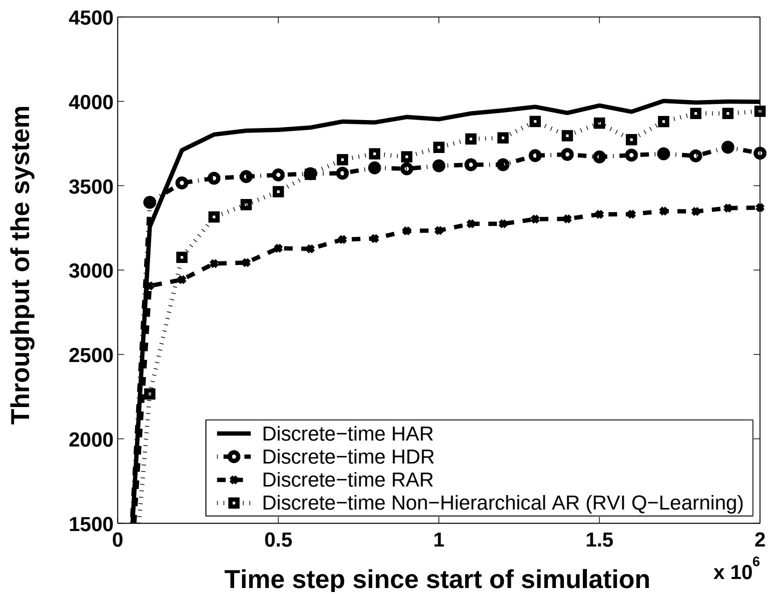


Figure 4.6. This plot shows that the discrete-time HAR algorithm performs better than the discounted reward HDR and RAR algorithms on the AGV scheduling task. It also demonstrates the faster convergence of the HAR algorithm comparing to RVI Q-learning, the non-hierarchical average reward algorithm.

algorithm. This figure also shows that the HAR algorithm converges faster to the same throughput as the non-hierarchical average reward algorithm. The non-hierarchical average reward algorithm used in this experiment is a continuous-time version of the relative value iteration (RVI) Q-learning (Abounadi et al., 2001). The difference in convergence speed between flat and hierarchical algorithms becomes more significant as we increase the number of states.

These results are consistent with the hypothesis that the average reward framework is superior to the discounted framework for learning continuing tasks, such as queuing, scheduling, and flexible manufacturing. Moreover, average reward methods do not need careful tuning of the discount factor to find gain-optimal policies.

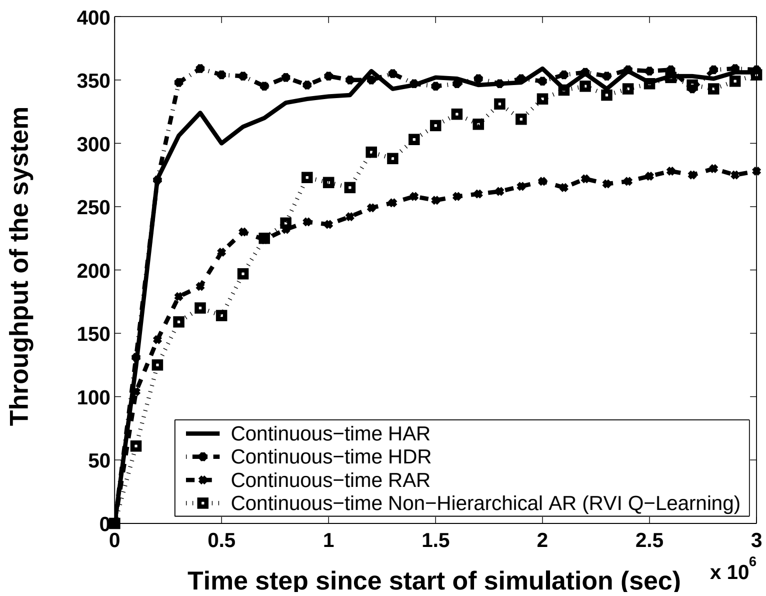


Figure 4.7. This plot shows that the continuous-time HAR converges to the same performance as the discounted reward HDR, and both outperform the recursively optimal average reward (RAR) algorithm on the AGV scheduling task. It also demonstrates the faster convergence of the HAR algorithm comparing to RVI Q-learning, the flat average reward algorithm.

4.4 Summary and Future Work

This chapter presents new discrete-time and continuous-time *hierarchically optimal average reward RL* (HAR) and *recursively optimal average reward RL* (RAR) algorithms applicable to continuing tasks, including manufacturing, scheduling, queuing, and inventory control. These algorithms are based on the average-reward SMDP model, which has been shown to be more appropriate for a wide class of continuing tasks than the better studied discounted reward SMDP model. *Hierarchically optimal average reward RL* (HAR) algorithms aim to find a hierarchical policy within the space of policies defined by the hierarchical decomposition that maximizes the *global gain*. In the *recursively optimal average reward RL* setting, the formulation of learning algorithms directly depends on the local optimality criterion used for each subtask in the hierarchy. The *recursively optimal average reward RL* (RAR) algorithms proposed in this chapter treat subtasks as continuing average reward problems and solve them by maximizing their gain given the policies of their children. We investigate the conditions under which the policy learned by the RAR algorithm at each subtask is independent of the context in which it is executed and therefore can be reused by other hierarchies. The effectiveness of the proposed algorithms were tested using two AGV scheduling tasks.

There are a number of directions for future work. An immediate question that arises is proving the asymptotic convergence of the algorithms to hierarchically optimal policies. These results should provide some theoretical validity to the proposed algorithms, in addition to their empirical effectiveness demonstrated in this chapter. Studying other local optimality criteria for subtasks in the hierarchy is an interesting problem that needs to be addressed. It helps to develop more effective *recursively optimal average reward RL* algorithms. It is also obvious that many other manufacturing and robotics problems can benefit from the algorithms proposed in this chapter.

CHAPTER 5

HIERARCHICAL POLICY GRADIENT REINFORCEMENT LEARNING

We illustrated value function (VF) and policy gradient (PG) solutions for MDPs in Section 2.2.4. As we described in that section, there are only weak theoretical guarantees on the performance of the value function reinforcement learning (VFRL) methods on problems with large discrete or continuous state spaces. We also mentioned that policy gradient reinforcement learning (PGRL) algorithms have received recent attention as a means to solve problems with continuous state spaces. They have also shown better performance when states are hidden. However, they are usually slower than VFRL methods. A possible solution is to incorporate prior knowledge and decompose the high-dimensional task into a collection of modules with smaller state spaces and learn these modules in a way to solve the overall problem. Hierarchical VFRL methods (Parr, 1998; Sutton et al., 1999; Dietterich, 2000; Andre and Russell, 2001) have been developed using this approach, as an attempt to scale RL to large state spaces.

In this chapter,[1] we propose a family of **hierarchical policy gradient reinforcement learning** (HPGRL) algorithms for scaling PGRL methods to problems with continuous (or large discrete) state and/or action spaces. In HPGRL, *non-primitive* subtasks are defined as PGRL problems. Later in this chapter, we accelerate learning in HPGRL algorithms by formulating high-level subtasks, which usually involve smaller state and finite action spaces, as VFRL problems, and low-level subtasks with infinite state and/or action spaces

[1] Most of the work presented in this chapter first appeared in Ghavamzadeh and Mahadevan (2003), "Hierarchical policy gradient algorithms," Proceedings of the Twentieth International Conference on Machine Learning, pp. 226-233.

as PGRL problems. This idea is similar to the idea used by Morimoto and Doya (2001) to learn stand-up behavior in a three-link, two-joint robot. We call this family of algorithms **hierarchical hybrid** algorithms.

The rest of this chapter is organized as follows. In Section 5.1, we describe how we define each subtask in a hierarchy as a PGRL problem. In Section 5.2, we introduce a family of HPGRL algorithms and compare the performance of this family of algorithms with a hierarchical VFRL algorithm and a flat RL algorithm in a simple taxi-fuel problem. In Section 5.3, we propose a family of *hierarchical hybrid* algorithms to accelerate learning in HPGRL algorithms. We illustrate this family of algorithms and demonstrate its performance using a continuous state and action ship steering problem. Finally, Section 5.4 summarizes the chapter and discusses some directions for future work.

5.1 Policy Gradient Formulation

In this section, we demonstrate how to define a subtask in a hierarchical task decomposition as a PGRL problem. We formulate a subtask in terms of a parameterized family of policies and a performance function. We then define a method to estimate the gradient of the performance function and a routine to update the policy parameters using this gradient. Our focus in this chapter is on episodic problems, so we assume that the overall task (*root* of the hierarchy) is episodic.

5.1.1 Policy Formulation

Each subtask M_i is defined using a set of randomized stationary policies $\mu_i(\boldsymbol{\theta}_i)$ parameterized in terms of a parameter vector $\boldsymbol{\theta}_i \in \mathbb{R}^K$. The term $\mu_i(a|s;\boldsymbol{\theta}_i)$ denotes the probability of taking action a in state s under the policy corresponding to $\boldsymbol{\theta}_i$. These parameterized policies for individual subtasks define a set of parameterized hierarchical policies $\boldsymbol{\mu}(\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is the vector of all subtasks' parameters. For every subtask M_i in the hierarchy, we make the following assumption about its set of parameterized policies $\mu_i(\boldsymbol{\theta}_i)$.

Assumption 5.1: For every state $s \in S_i$ and every action $a \in A_i$, $\mu_i(a|s;\boldsymbol{\theta}_i)$ as a function of $\boldsymbol{\theta}_i$, is bounded and has bounded first and second derivatives. Furthermore, $\frac{\nabla \mu_i(a|s;\boldsymbol{\theta}_i)}{\mu_i(a|s;\boldsymbol{\theta}_i)}$ is bounded, differentiable, and has bounded first derivatives. $\square$

In HRL methods, we typically assume that every time a subtask M_i is called, it starts at one of its initial states ($\in \mathcal{I}_i$) and terminates at one of its terminal states ($\in T_i$) after a finite number of steps. Therefore, we make the following assumption for every subtask M_i in the hierarchy. Under this assumption, each subtask can be considered an episodic problem and each instantiation of a subtask can be considered an episode.

Assumption 5.2 (Subtask Termination): We define a dummy state $s_i^* \in S_i$ such that, for every action $a \in A_i$ and every terminal state s_{T_i}, we have

$$r_i(s_{T_i}, a) = 0 \quad \text{and} \quad P_i(s_i^*, 1|s_{T_i}, a) = 1$$
$$r_i(s_i^*, a) = 0 \quad \text{and} \quad P_i(s_i^*, 1|s_i^*, a) = 1$$

and for all hierarchical stationary policies $\boldsymbol{\mu}(\boldsymbol{\theta})$ and non-terminal states $s \in S_i$, we have

$$F_i^{\boldsymbol{\mu}(\boldsymbol{\theta})}(s_i^*, 1|s) = 0$$

and finally for all states $s \in S_i$, we have

$$F_i^{\boldsymbol{\mu}(\boldsymbol{\theta})}(s_i^*, N|s) > 0$$

where $F_i^{\boldsymbol{\mu}(\boldsymbol{\theta})}$ is the multi-step abstract transition probability function of subtask M_i under the hierarchical policy $\boldsymbol{\mu}(\boldsymbol{\theta})$ described in Section 3.2, and $N = |S_i|$ is the number of states

in the state space of subtask M_i. $\square$

Under this assumption, all terminal states of subtask M_i transition with probability 1 and reward 0 to the dummy state s_i^* and stay there until the next instantiation of subtask M_i as shown in Figure 5.1. This is a dummy transition and does not add another time-step to the cycle of subtask M_i.

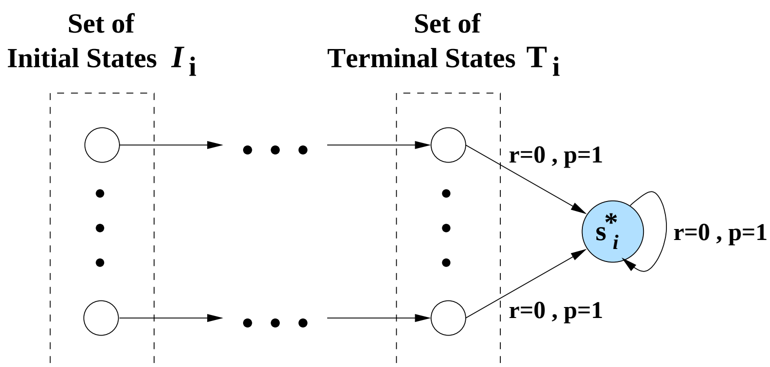


Figure 5.1. This figure shows how we model a subtask as an *episodic* problem under Assumption 5.2.

Under this model, for every hierarchical policy $\boldsymbol{\mu}(\boldsymbol{\theta})$, we define a new MDP M_{I_i} for each subtask M_i with abstract transition probabilities and rewards

$$F_{I_i}^{\boldsymbol{\mu}(\boldsymbol{\theta})}(s',1|s) = \begin{cases} F_i^{\boldsymbol{\mu}(\boldsymbol{\theta})}(s',1|s) & s \neq s_i^*, \\ I_i(s') & s = s_i^*. \end{cases} \tag{5.1}$$

$$r_{I_i}(s,a;\boldsymbol{\theta}) = r_i(s,a;\boldsymbol{\theta})$$

where $I_i(s)$ is the probability that subtask M_i starts at state s.

Let $\mathcal{F}_{I_i}^{\boldsymbol{\mu}(\boldsymbol{\theta})}$ be the set of all abstract transition probability functions $F_{I_i}^{\boldsymbol{\mu}(\boldsymbol{\theta})}$. We have the following result for subtask M_i.

Lemma 5.1: Let Assumptions 5.1 and 5.2 hold. Then for every $F_{I_i}^{\boldsymbol{\mu}(\boldsymbol{\theta})} \in \mathcal{F}_{I_i}^{\boldsymbol{\mu}(\boldsymbol{\theta})}$ and every state $s \in S_i$, we have $\sum_{N=1}^{|S_i|} F_{I_i}^{\boldsymbol{\mu}(\boldsymbol{\theta})}(s_i^*, N|s) > 0$. $\square$

Lemma 5.1 is equivalent to assuming that the MDP M_{I_i} is recurrent, i.e., the underlying Markov chain for every policy $\boldsymbol{\mu}(\boldsymbol{\theta})$ in this MDP has a single recurrent class and the state s_i^* is a recurrent state. In this case, the balance equations

$$\sum_{s=1}^{|S_i|} F_{I_i}^{\boldsymbol{\mu}(\boldsymbol{\theta})}(s', 1|s)\pi_i(s) = \pi_i(s'), \qquad \forall s' \in S_i \quad , \quad s' \neq s$$

$$\sum_{s=1}^{|S_i|} \pi_i(s) = 1$$

have a unique solution $\pi_{I_i}^{\boldsymbol{\mu}(\boldsymbol{\theta})}$. We refer to $\pi_{I_i}^{\boldsymbol{\mu}(\boldsymbol{\theta})}$ as the steady state probability vector of the Markov chain with transition probabilities defined by Equation 5.1, and to $\pi_{I_i}^{\boldsymbol{\mu}(\boldsymbol{\theta})}(s)$ as the steady state probability of being in state s.

5.1.2 Performance Measure Definition and Optimization

We define **weighted reward-to-go**, $\chi_i(\boldsymbol{\theta})$, as the performance measure of subtask M_i under the parameterized hierarchical policy $\boldsymbol{\mu}(\boldsymbol{\theta})$, and for which Assumption 5.2 holds, as

$$\chi_i(\boldsymbol{\theta}) = \sum_{s \in S_i} I_i(s) J_i(s; \boldsymbol{\theta})$$

The term $J_i(s; \boldsymbol{\theta})$ is the reward-to-go of subtask M_i in state s under hierarchical policy $\boldsymbol{\mu}(\boldsymbol{\theta})$ and is defined as

$$J_i(s; \boldsymbol{\theta}) = E\left[\sum_{k=0}^{T-1} r_i(s_k, a_k) | s_0 = s; \boldsymbol{\theta}\right]$$

where $T = \min\{k > 0 | s_k = s_i^*\}$ is the first future time that state s_i^* is visited.[2]

[2]With the definition of absorbing state s_i^* in our model (see Figure 5.1), the reward-to-go of subtask M_i in state s, $J_i(s; \boldsymbol{\theta})$, is the same as undiscounted projected value function of subtask M_i in state s.

In order to obtain an expression for the gradient $\nabla \chi_i(\boldsymbol{\theta})$, we use MDP M_{I_i} defined in Section 5.1.1. Using Lemma 5.1, MDP M_{I_i} is recurrent. For MDP M_{I_i}, let $\pi_{I_i}^{\boldsymbol{\mu}(\boldsymbol{\theta})}(s)$ be the steady state probability distribution of being in state s at subtask M_i and let $E_{I_i}[T|\boldsymbol{\theta}]$ be the mean recurrence time of subtask M_i, i.e., $E_{I_i}[T|\boldsymbol{\theta}] = E_{I_i}[T|s_0 = s_i^*; \boldsymbol{\theta}]$, under the hierarchical policy $\boldsymbol{\mu}(\boldsymbol{\theta})$. We also define $\tilde{J}_i(s, a; \boldsymbol{\theta})$[3] as

$$\tilde{J}_i(s, a; \boldsymbol{\theta}) = E_{I_i}\left[\sum_{k=0}^{T-1} r_{I_i}(s_k, a_k)|s_0 = s, a_0 = a; \boldsymbol{\theta}\right]$$

Using recurrent MDP M_{I_i}, we can derive the following proposition which gives an expression for the gradient of the weighted reward-to-go $\chi_i(\boldsymbol{\theta})$ with respect to the parameter vector $\boldsymbol{\theta}$.

Proposition 5.1: If Assumptions 5.1 and 5.2 hold

$$\nabla \chi_i(\boldsymbol{\theta}) = E_{I_i}[T|\boldsymbol{\theta}] \sum_{s \in S_i} \sum_{a \in A_i} \pi_{I_i}^{\boldsymbol{\mu}(\boldsymbol{\theta})}(s) \nabla \mu_i(a|s; \boldsymbol{\theta}_i) \tilde{J}_i(s, a; \boldsymbol{\theta})$$

□

This proposition is similar to Proposition 1 on page 35 of Marbach (1998).

The expression for the gradient in Proposition 5.1 can be estimated over a **renewal cycle** (cycle between consecutive visits to recurrent state s_i^*) as

$$F_{m,i}(\boldsymbol{\theta}) = \sum_{n=t_m}^{t_{m+1}-1} R_i(s_n, a_n; \boldsymbol{\theta}) \frac{\nabla \mu_i(s_n, a_n; \boldsymbol{\theta}_i)}{\mu_i(s_n, a_n; \boldsymbol{\theta}_i)} \tag{5.2}$$

where t_m is the time of the mth visit at the recurrent state s_i^* and $R_i(s_n, a_n; \boldsymbol{\theta}) = \sum_{k=n}^{t_{m+1}-1} r_i(s_n, a_n; \boldsymbol{\theta})$ is an estimate of $\tilde{J}_i(s_n, a_n; \boldsymbol{\theta})$.

[3] With the defi nition of absorbing state in Figure 5.1, $\tilde{J}_i$ is the undiscounted projected action-value function of subtask M_i.

From Equation 5.2, we obtain the following procedure to update $\boldsymbol{\theta}_i$, the parameter vector of subtask M_i, along the approximate gradient direction at every time step.

$$z_{k+1,i} = \begin{cases} 0 & s_k = s_i^*, \\ z_{k,i} + \frac{\nabla \mu_i(a_k|s_k;\boldsymbol{\theta}_{k,i})}{\mu_i(a_k|s_k;\boldsymbol{\theta}_{k,i})} & \text{otherwise.} \end{cases}$$

$$\boldsymbol{\theta}_{k+1,i} = \boldsymbol{\theta}_{k,i} + \alpha_{k,i} R_i(s_k, a_k; \boldsymbol{\theta}_k) z_{k+1,i} \tag{5.3}$$

where $\alpha_{k,i}$ is the step size parameter for subtask M_i and satisfies the following assumptions.

Assumption 5.3: $\alpha_{k,i}$'s are deterministic, nonnegative, and satisfy $\sum_{k=1}^{\infty} \alpha_{k,i} = \infty$ and $\sum_{k=1}^{\infty} \alpha_{k,i}^2 < \infty$. □

Assumption 5.4: $\alpha_{k,i}$'s are non-increasing and there exists a positive integer p and a positive scalar A such that $\sum_{k=n}^{n+t} (\alpha_{n,i} - \alpha_{k,i}) \leq A t^p \alpha_{n,i}^2$ for all positive integers n and t. □

We have the following convergence result for the iterative procedure in Equation 5.3 to update the parameters.

Proposition 5.2: Let Assumptions 5.1, 5.2, 5.3, and 5.4 hold, and let $\boldsymbol{\theta}_k$ be the sequence of parameter vectors generated by Equation 5.3. Then, the estimation of performance measure $\chi_i(\boldsymbol{\theta}_k)$ converges and $\lim_{k\to\infty} \nabla \chi_i(\boldsymbol{\theta}_k) = 0$ with probability 1. □

This proposition is similar to Proposition 14 on page 59 of Marbach (1998).

Equation 5.3 provides an unbiased estimate of $\nabla \chi_i(\boldsymbol{\theta})$. For systems involving a large state space, the interval between visits to state s_i^* can be large. As a consequence, the estimate of $\nabla \chi_i(\boldsymbol{\theta})$ might have a large variance. Several methods have been proposed to reduce the variance in this estimation and yield faster convergence (Marbach, 1998;

Baxter and Bartlett, 2001). For instance, we can use a discount factor γ in the reward-to-go estimation. However, these methods introduce a bias into the estimate of $\nabla\chi_i(\boldsymbol{\theta})$. For these methods, we can derive a modified version of Equation 5.3 to incrementally update the parameter vector along the approximate gradient direction.

5.2 Hierarchical Policy Gradient Algorithms

After decomposing the overall task to a set of subtasks as described in Chapter 3, and formulating each subtask in the hierarchy as an episodic PGRL problem as illustrated in Section 5.1, we can use the update Equation 5.3 and derive an HPGRL algorithm to maximize the weighted reward-to-go for every subtask in the hierarchy. Algorithm 3 shows the pseudo code for this algorithm.

Algorithm 3 A hierarchical policy gradient algorithm that maximizes the weighted reward-to-go for the subtasks in the hierarchy.

Function HPGRL(Task M_i**, State** s**)**
$RR = 0$
if M_i is a primitive action **then**
    execute action i in state s, observe state s' and reward $r(s,i)$
    return $r(s,i)$
else /* M_i is a non-primitive subtask */
    while M_i has not terminated ($s \neq s_i^*$) **do**
        choose action a using policy $\mu_i(s;\boldsymbol{\theta}_i)$
        R=**HPGRL(Task** M_a**, State** s**)**
        observe result state s' and internal reward $\tilde{r}_i(s,a)$
        if $s' = s_i^*$ **then**
            $z_{k+1,i} = 0$
        else
            $z_{k+1,i} = z_{k,i} + \frac{\nabla\mu_i(a|s;\boldsymbol{\theta}_{k,i})}{\mu_i(a|s;\boldsymbol{\theta}_{k,i})}$
        end if
        $\boldsymbol{\theta}_{k+1,i} = \boldsymbol{\theta}_{k,i} + \alpha_{k,i}\left[R + \tilde{r}_i(s,a)\right] z_{k+1,i}$
        $RR = RR + R$
        $s = s'$
    end while
end if
return RR
end HPGRL

The term $\tilde{r}_i(s, a)$ on Lines 10 and 16 of the algorithm is the internal reward which can be used only inside each subtask to speed up its local learning and does not propagate to the upper levels in the hierarchy. Lines 11 to 16 can be replaced with any other policy gradient algorithm to optimize weighted reward-to-go, such as those presented in Marbach (1998) or Baxter and Bartlett (2001). Thus, Algorithm 3 describes a family of HPGRL algorithms to maximize the weighted reward-to-go for every subtask in the hierarchy.

The above formulation of each subtask brings the following limitations for the learned policy: **1)** Parameterized representation of a policy limits the policy search to a set which is typically smaller than the set of all possible policies. **2)** Gradient-based policy search methods find a solution which is locally, rather than globally, optimal. Thus, in general, the family of algorithms described above converges to a **recursively local optimal** policy. If the policy learned for every subtask in the hierarchy coincides with the best policies, then these algorithms converge to a *recursively optimal policy*.

5.2.1 Taxi-Fuel Problem

In this section, we apply the HPGRL algorithm to the taxi-fuel problem introduced in Dietterich (1998), and compare its performance with MAXQ-Q, a value function hierarchical RL algorithm (Dietterich, 2000), and flat Q-learning.

A 5-by-5 grid world inhabited by a taxi is shown in Figure 5.2. There are four stations marked as B(lue), G(reen), R(ed), and Y(ellow). The task is episodic. In each episode, the taxi starts in a randomly chosen location and with a randomly chosen amount of fuel ranging from 5 to 12 units. There is a passenger at one of the four stations (chosen randomly), and that passenger wishes to be transported to one of the other three stations (also chosen randomly). The taxi must go to the passenger's location, pick up the passenger, go to its destination location and drop off the passenger there. The episode ends when the passenger is deposited at its destination station or taxi goes out of fuel. There are $8,750$ possible states and 7 primitive actions in the domain, *Pickup*, *Dropoff*, *Fillup*, and four *navigation* actions

(each of these consumes one unit of fuel). Each action is deterministic. There is a reward of -1 for each action and an additional reward of 20 for successfully delivering the passenger. There is a reward of -10 if the taxi attempts to execute the *Dropoff* or *Pickup* actions illegally, and a reward of -20 if the fuel level falls below zero. The system performance is measured in terms of the average reward per step which is equivalent to maximizing the total reward per episode in this task. Each experiment was conducted ten times and the results averaged.

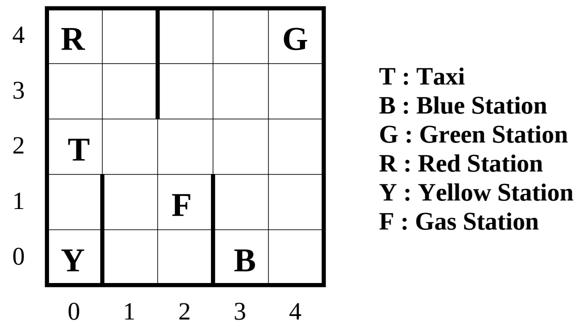


Figure 5.2. The taxi-fuel problem.

Figure 5.3 compares the performance of HPGRL, MAXQ-Q and flat Q-learning algorithms on the taxi-fuel problem.[4] The hierarchical policy gradient algorithm used in this experiment is the one shown in Algorithm 3, with one policy parameter for each state-action pair (s, a). The graph shows that MAXQ-Q converges faster than HPGRL and flat Q-learning, and HPGRL is slightly faster than flat Q-learning.

As we expected, the HPGRL algorithm converges to the same performance as MAXQ-Q. However, it is much slower than its value function based counterpart. The performance of HPGRL can be improved by better policy formulation and using more sophisticated policy gradient algorithms for each subtask. The slow convergence of HPGRL algorithms

[4]Both HPGRL and MAXQ-Q utilize the hierarchical task decomposition used in Dietterich (1998).

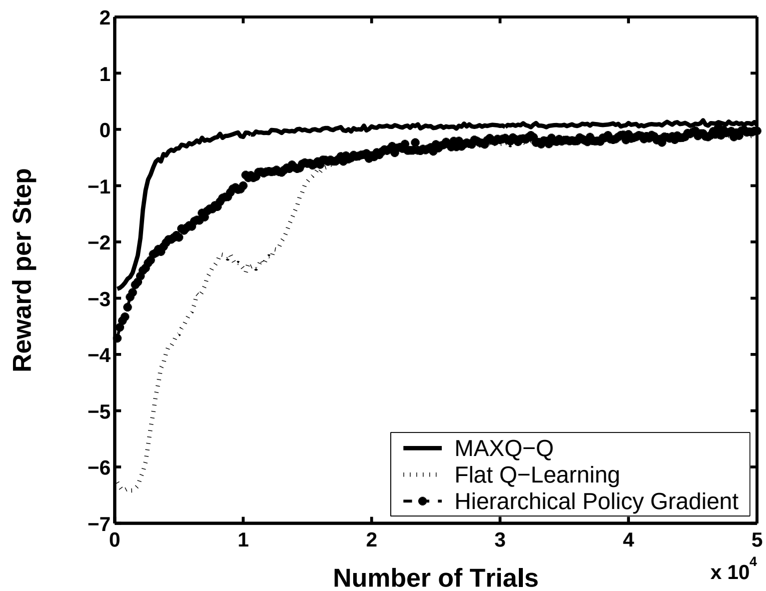


Figure 5.3. This figure compares the performance of the HPGRL algorithm proposed in this section with MAXQ-Q and flat Q-learning algorithms on the taxi-fuel problem.

motivates us to use both VFRL and PGRL methods in a hierarchy. We address this by introducing *hierarchical hybrid* algorithms in the next section.

5.3 Hierarchical Hybrid Algorithms

Despite the methods proposed to reduce the variance of gradient estimators in PGRL algorithms, these algorithms are still slower than VFRL methods as shown in the simple taxi-fuel experiment in Section 5.2.1. We accelerate learning of HPGRL algorithms by formulating those subtasks with smaller state spaces and finite action spaces usually located at the high levels of the hierarchy as VFRL problems, and those with large state spaces and/or infinite action spaces usually located at the low levels of the hierarchy as PGRL problems. This formulation can benefit from the faster convergence of VFRL methods and the power of PGRL algorithms in domains with infinite state and/or action spaces at the same time. We call this family of algorithms, **hierarchical hybrid** algorithms and illustrate them using a ship steering task.

Figure 5.4 shows a ship steering task (Miller et al., 1990). A ship starts at a randomly chosen position, orientation, and turning rate. Its goal is to be maneuvered at a constant speed through a gate placed at a fixed position. The ship does not know the location of the gate and observes the gate only when it passes through it.

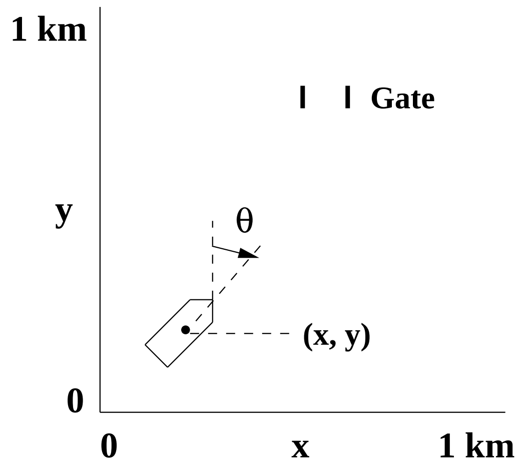


Figure 5.4. The ship steering task.

Equations 5.4 gives the motion equations of the ship, where $T = 5$ is the time constant of convergence to desired turning rate, $V = 3\ m/sec$ is the constant speed of the ship, and $\Delta = 0.2\ sec$ is the sampling interval. There is a time lag between changes in the desired turning rate and the actual turning rate, modeling the effects of a real ship's inertia and the resistance of the water.

$$\begin{aligned} x[t+1] &= x[t] + \Delta V \sin\theta[t] \\ y[t+1] &= y[t] + \Delta V \cos\theta[t] \\ \theta[t+1] &= \theta[t] + \Delta\dot{\theta}[t] \\ \dot{\theta}[t+1] &= \dot{\theta}[t] + \Delta(r[t] - \dot{\theta}[t])/T \end{aligned} \tag{5.4}$$

At each time t, the state of the ship is given by its position $x[t]$ and $y[t]$, orientation $\theta[t]$ and actual turning rate $\dot{\theta}[t]$. The action is the desired turning rate of the ship $r[t]$. All

State	x	0 to 1000 meters
	y	0 to 1000 meters
	θ	-180 to 180 degrees
	$\dot{\theta}$	-15 to 15 degrees/sec
Action	r	-15 to 15 degrees/sec

Table 5.1. State and action variables for the ship steering task.

four state variables and also the action are continuous and their range is shown in Table 5.1. The ship steering problem is episodic. In each episode, the goal is learning to generate sequences of actions that steer the center of the ship through the gate in the minimum amount of time. The sides of the gate are placed at coordinates (350,400) and (450,400). If the ship moves out of bound ($x < 0$ or $x > 1000$ or $y < 0$ or $y > 1000$), the episode terminates and is considered as a failure.

We applied both a flat PGRL algorithm and an actor-critic algorithm (Konda, 2002) to this task without achieving a good performance in a reasonable amount of time. Figure 5.7 shows that after learning for $50,000$ episodes, these algorithms are able to control the ship to successfully pass through the gate only 60 percent of time. We believe this occurred due to two reasons, which make this problem hard to learn. First, since the ship cannot turn faster than 15 degrees/sec, all state variables change only by a small amount at each control interval. Thus, we need a high resolution discretization of the state space in order to accurately model state transitions, which requires a large number of parameters for the function approximator and makes the problem intractable. Second, there is a time lag between changes in the desired turning rate r and the actual turning rate $\dot{\theta}$, ship's position x, y, and orientation θ, which requires the controller to deal with long delays.

However, we successfully applied a flat policy gradient algorithm to the simplified versions of this problem shown in Figure 5.5, when x and y change from 0 to 150 instead of 0 to 1000, the ship always starts at a fixed position (initial positions in Figure 5.5) with randomly chosen orientation and turning rate, and the goal is to reach to a neighborhood of a pre-defined point (goals in Figure 5.5). It indicates that this high-dimensional non-linear

control problem can be learned using an appropriate hierarchical decomposition. Using this prior knowledge, we decompose the problem into two levels using the task graph shown in Figure 5.6. At the high-level, the agent learns to select among four diagonal and four horizontal/vertical subtasks. At the low-level, each low-level subtask learns a sequence of turning rates to achieve its own goal. We use symmetry and map eight subtasks located below the *root* to only two subtasks at the low-level, one associated with four diagonal subtasks and one associated with four horizontal/vertical subtasks as shown in Figure 5.6. We call them *diagonal* and *horizontal/vertical* subtasks.

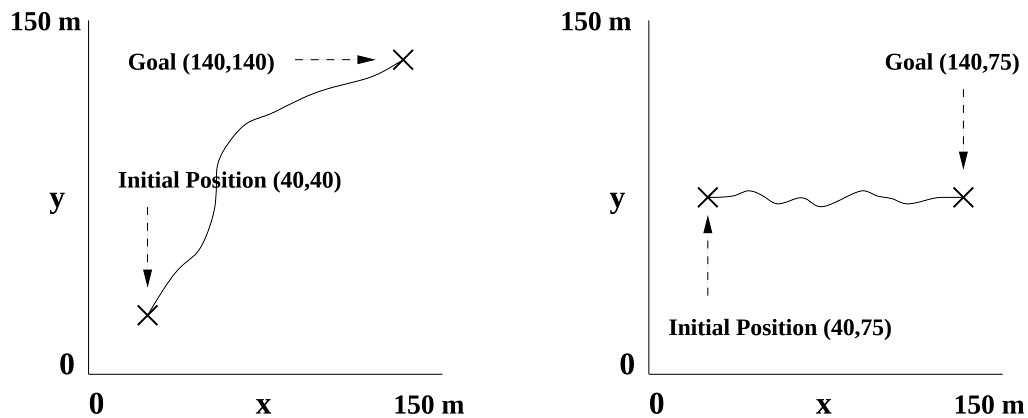


Figure 5.5. This figure shows two simplified versions of the ship steering task used as low-level subtasks in the hierarchical decomposition of the ship steering problem.

The flat PGRL algorithm used in this section uses Equation 5.3 and CMAC function approximator with 9 four-dimensional tilings, dividing the space into $20 \times 20 \times 36 \times 5 = 72,000$ tiles each. The actor-critic algorithm also uses the above function approximator for its actor, and 9 five dimensional tilings of size $5 \times 5 \times 36 \times 5 \times 30 = 135,000$ tiles for its critic. The fifth dimension of critic's tilings is for the continuous action.

In the *hierarchical hybrid* algorithm, we decompose the task using the task graph in Figure 5.6. At the high-level, the learner explores in a low-dimensional sub-space of the original high-dimensional state space. The state variables are only the coordinates of the ship x and y with the full range from 0 to 1000. The actions are four diagonal and four

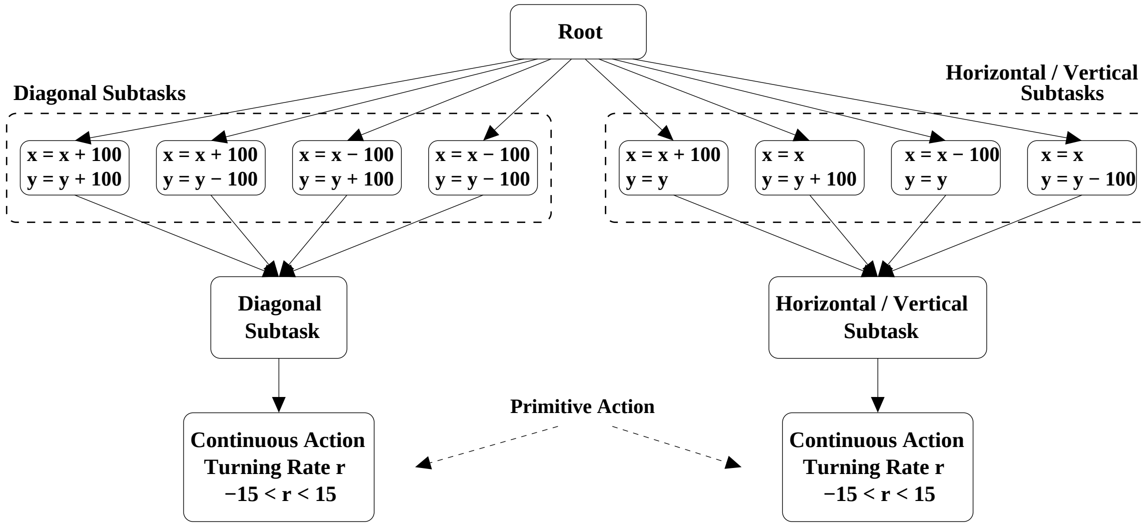


Figure 5.6. A task graph for the ship steering problem.

horizontal/vertical subtasks similar to those subtasks shown in Figure 5.5. The state space is coarsely discretized into 400 states. We use the value-based Q(λ) algorithm with ϵ-greedy action selection and replacing traces to learn a sequence of diagonal and horizontal/vertical subtasks to achieve the goal of the entire task (passing through the gate). Each episode ends when the ship passes through the gate or moves out of bound. Then the new episode starts with the ship in a randomly chosen position, orientation, and turning rate. In this algorithm, λ is set to 0.9, learning rate to 0.1, and ϵ starts with 0.1 remains unchanged until the performances of low-level subtasks reach to a certain level and then is decreased by a factor of 1.01 every 50 episodes.

At the low-level, the learner explores local areas of the high-dimensional state space without discretization. When the high-level learner selects one of the low-level subtasks, the low-level subtask takes control and executes the following steps as shown in Figure 5.5. **1)** Maps the ship to a new coordinate system in which the ship is in position $(40, 40)$ for the diagonal subtask and $(40, 75)$ for the horizontal/vertical subtask. **2)** Sets the low-level goal to position $(140, 140)$ for the diagonal subtask and $(140, 75)$ for the horizontal/vertical subtask. **3)** Sets the low-level boundaries to $0 \leq x\ ,\ y \leq 150$. **4)** Generates primitive actions

until either the ship reaches to a neighborhood of the low-level goal, a circle with radius 10 around the low-level goal (success), or moves out of the low-level bounds (failure).

The two low-level subtasks use all four state variables, however the range of coordination variables x and y is 0 to 150 instead of 0 to 1000. Their action variable is the desired turning rate of the ship, which is a continuous variable with range -15 to 15 $degrees/sec$. The control interval is 0.6 sec (three times the sampling interval $\Delta = 0.2$ sec). They use the PGRL algorithm on Lines 11 to 16 of Algorithm 3 to update their parameters. In addition, they use a CMAC function approximator with 9 four dimensional tilings, dividing the space into $5 \times 5 \times 36 \times 5 = 4,500$ tiles each. One parameter w is defined for each tile and the parameterized policy is a Gaussian:

$$\mu(s, a, W) = \frac{1}{\sqrt{2\pi}} e^{-\frac{A}{2}} \qquad , \qquad A = \frac{\sum_{i=0}^{N} w_i \phi_i}{\sum_{i=0}^{N} \phi_i}$$

where $N = 9 \times 4,500 = 40,500$ is the total number of tiles and ϕ_i is 1 if state s falls in tile i and 0 otherwise. The actual action is generated after mapping the value chosen by the Gaussian policy to the range from -15 to 15 $degrees/sec$ using a sigmoid function.

In addition to the original reward of -1 per step, we define internal rewards 100 and -100 for low-level success and failure, and a reward according to the distance of the current ship orientation θ to the angle between the current position and low-level goal $\hat{\theta}$ given by

$$G = exp\left(-\frac{\|\theta - \hat{\theta}\|^2}{30 \times 30}\right) - 1$$

where 30 $degrees$ gives the width of the reward function. When a low-level subtask terminates, the only reward that propagates to the high-level is the summation of all -1 rewards per step. In addition to reward received from low-level, high-level uses a reward 100 upon successfully passing through the gate.

We trained the system for $50,000$ episodes. In each episode, the high-level learner (controller located at *root*) selects a low-level subtask, and the selected low-level subtask

is executed until it successfully terminates (ship reaches the low-level goal) or it fails (ship goes out of the low-level bounds). Then control returns to the high-level subtask (*root*) again. The following results are averaged over five simulation runs.

Figure 5.7 compares the performance of the *hierarchical hybrid* algorithm with flat PGRL and actor-critic algorithms in terms of the number of successful trials in 1000 episodes. As this figure shows, despite the high resolution function approximators used in both flat algorithms, their performance is worse than the *hierarchical hybrid* algorithm. Moreover, their computation time per step is also much more than the *hierarchical hybrid* algorithm, due to the large number of parameters to be learned.

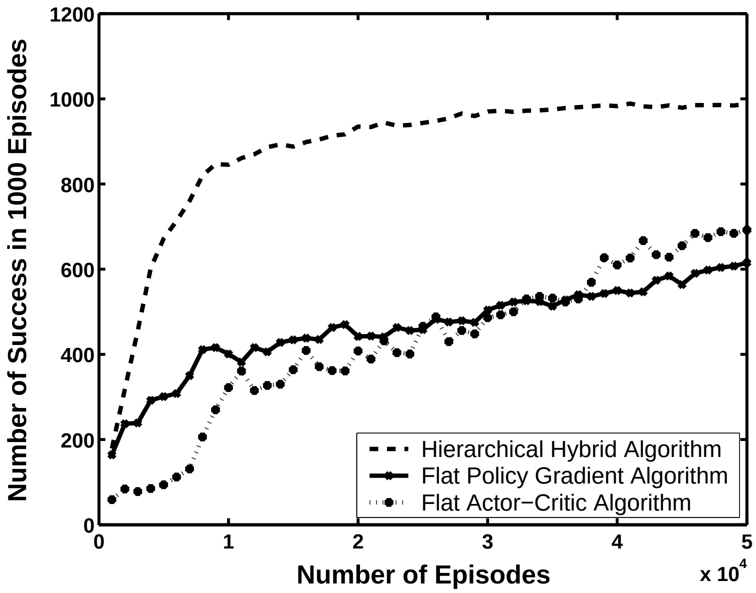


Figure 5.7. This figure shows the performance of *hierarchical hybrid*, flat PGRL and actor-critic algorithms in terms of the number of successful trials in 1000 episodes.

Figure 5.8 demonstrates the performance of the *hierarchical hybrid* algorithm in terms of the average number of low-level subtask calls. This figure shows that after learning, the learner executes about 4 low-level subtasks (diagonal or horizontal/vertical subtasks) per episode.

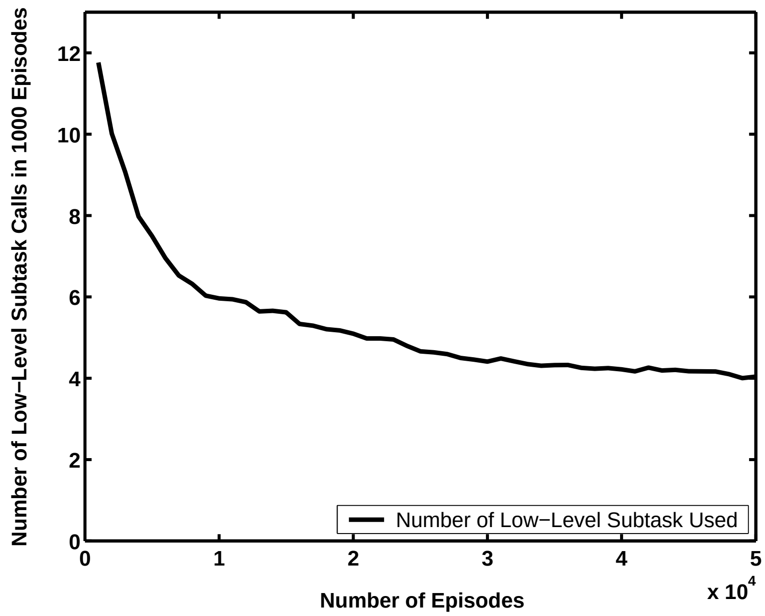


Figure 5.8. This figure shows the performance of the *hierarchical hybrid* algorithm in terms of the number of low-level subtask calls.

Figure 5.9 compares the performance of *hierarchical hybrid*, flat PGRL and actor-critic algorithms in terms of the average number of steps to goal (averaged over 1000 episodes). This figure shows that after learning, it takes about 220 primitive actions (turn actions) for the *hierarchical hybrid* learner to pass through the gate.

Figures 5.10 and 5.11 show the performance of the diagonal and horizontal/vertical subtasks in terms of the number of success out of 1000 executions respectively.

Finally, Figure 5.12 demonstrates the learned policy for two sample initial configurations of the ship shown with big circles. The upper configuration is $x = 700$, $y = 700$, $\theta = 100$, $\dot{\theta} = 3.65$ and the lower one is $x = 750$, $y = 180$, $\theta = 80$, $\dot{\theta} = 7.9$. The low-level subtasks chosen by the agent at the high-level are shown by small circles in this figure.

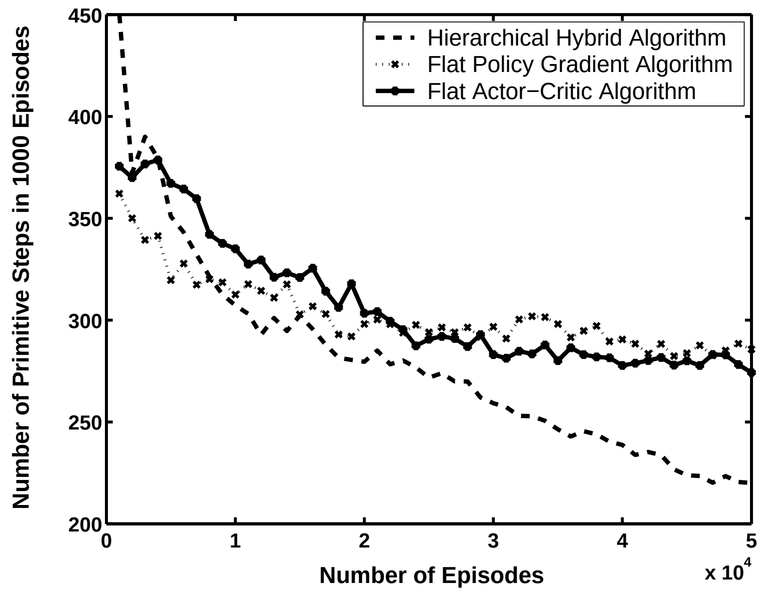


Figure 5.9. This figure shows the performance of *hierarchical hybrid*, flat PGRL and actor-critic algorithms in terms of the number of steps to pass through the gate.

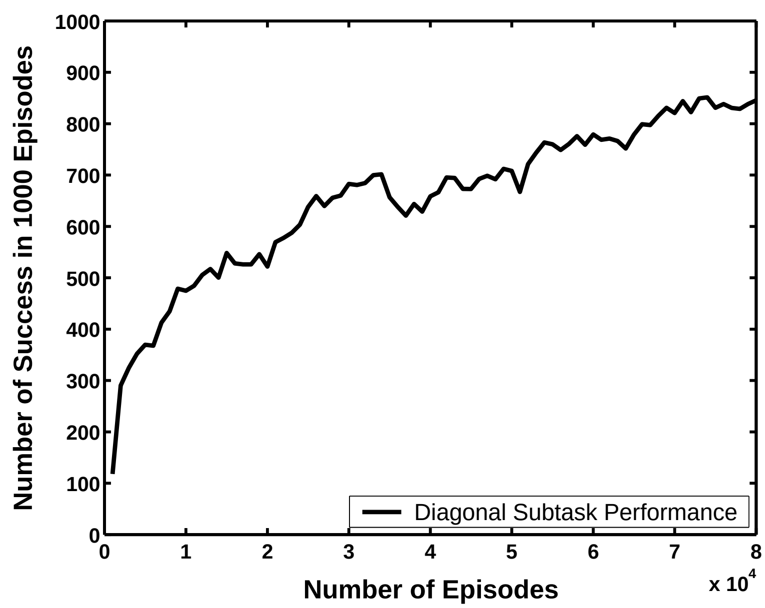


Figure 5.10. This figure shows the performance of the diagonal subtask in terms of the number of successful trials in 1000 episodes.

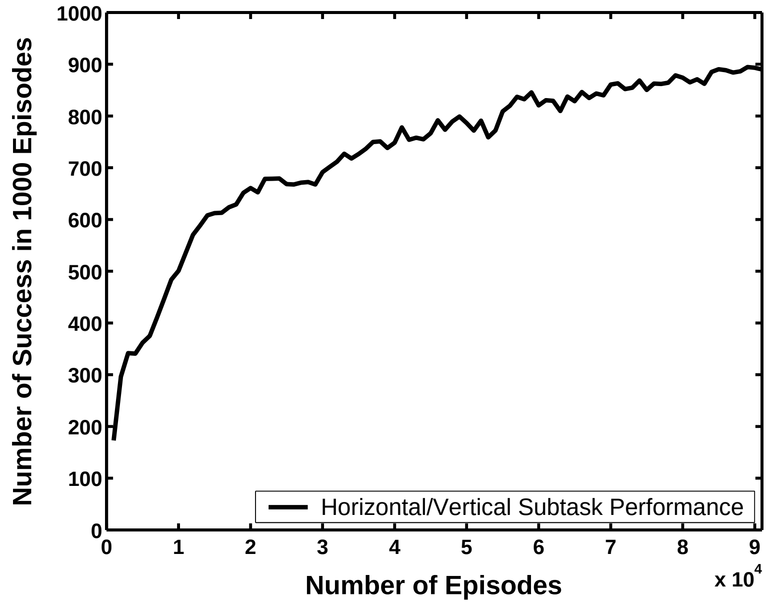


Figure 5.11. This figure shows the performance of the horizontal/vertical subtask in terms of the number of successful trials in 1000 episodes.

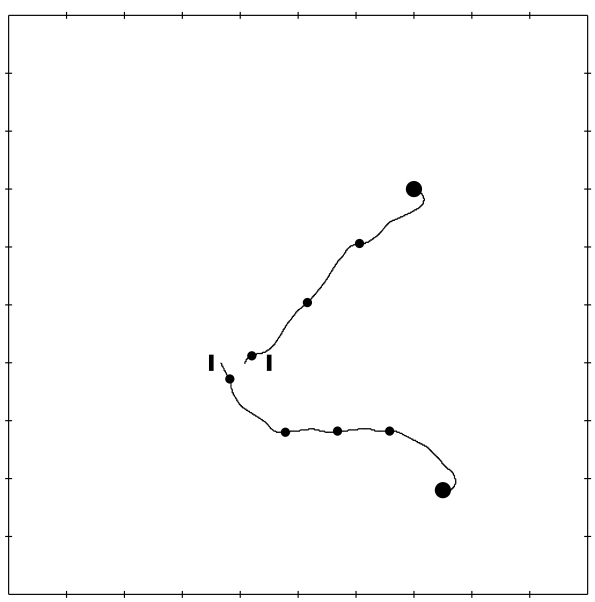


Figure 5.12. This figure shows the learned policy for two initial configurations of the ship.

5.4 Summary and Future Work

In this chapter, we described HPGRL, a family of *hierarchical policy gradient RL* algorithms for learning in domains with continuous state and/or action spaces. We compared the performance of this family of algorithms with a hierarchical VFRL algorithm and a flat RL algorithm in a simple taxi-fuel problem. The results demonstrate that the HPGRL algorithm converges slower than the hierarchical VFRL algorithm. To accelerate learning in HPGRL algorithms, we proposed a family of *hierarchical hybrid* algorithms in which subtasks located at high level(s) of the hierarchy are formulated as VFRL, and subtasks located at low level(s) of the hierarchy are defined as PGRL problems. We use a continuous state and action ship steering task to illustrate this family of algorithms and to demonstrate their performance.

The algorithms proposed in this chapter are based on the assumption that the overall task (*root* of the hierarchy) is *episodic*. One direction for future work is to reformulate the algorithms presented in this chapter for the case when the overall task is *continuing*. In this case, the *root* task is formulated as a *continuing* problem with the *average reward* as its performance function. Since the policy learned at *root* involves policies of its children, the type of optimality achieved at *root* depends on how we formulate other subtasks in the hierarchy. Different notions of optimality in *hierarchical average reward* presented in Chapter 4 can be used to develop new HPGRL algorithms for *continuing* problems.

Although the proposed algorithms give us the ability to deal with continuous state and continuous action spaces, they are not still appropriate to efficiently control real-world problems in which the speed of learning is crucial. The results of ship steering task indicate that in order to apply the proposed algorithms to real-world domains, more powerful PGRL algorithms are needed to be developed — PGRL algorithms that need a smaller number of samples to learn a good policy, and are less computationally expensive.

CHAPTER 6

HIERARCHICAL MULTI-AGENT REINFORCEMENT LEARNING

In this chapter,[1] we investigate the use of hierarchical reinforcement learning (HRL) to speed up the acquisition of cooperative multi-agent tasks. Our approach to learning in cooperative multi-agent domains differs from all the approaches discussed in Section 2.5 in one key respect, namely the use of hierarchy to speed up multi-agent reinforcement learning. The key idea underlying our approach is that coordination skills are learned much more efficiently if the agents have a hierarchical representation of the task structure. Algorithms for learning task-level coordination have also been developed in non-MDP approaches, see Sugawara and Lesser (1998). We first introduce a hierarchical multi-agent RL framework. In this framework, we assume agents are cooperative and each agent is given an initial hierarchical decomposition of the overall task. Moreover, agents are *homogeneous,* i.e., use the same hierarchical task decomposition. However, learning is decentralized, with each agent learning three interrelated skills: how to perform subtasks, which order to do them in, and how to coordinate with other agents. The use of hierarchy speeds up learning in multi-agent domains by making it possible to learn coordination skills at the level of subtasks instead of primitive actions. We define *cooperative subtasks* to be those subtasks in which coordination among agents significantly improves the performance of the over-

[1] Most of the work presented in this chapter first appeared in **1)** Makar, Mahadevan and Ghavamzadeh (2001), "Hierarchical multi-agent reinforcement learning," Proceedings of the Fifth International Conference on Autonomous Agents, pp. 246-253, and **2)** Ghavamzadeh and Mahadevan (2004), "Learning to Communicate and Act using Hierarchical Reinforcement Learning," Proceedings of the Third International Joint Conference on Autonomous Agents and Multi-Agent Systems, pp. 1114-1121. A longer version of this work has also been submitted to the Journal of Autonomous Agents and Multi-Agent Systems.

all task. Agents cooperate with their teammates at *cooperative subtasks* and ignore them while performing *non-cooperative* subtasks. Those levels of the hierarchy which include *cooperative subtasks* are called *cooperation levels*. Since high-level coordination allows for increased cooperation skills as agents do not get confused by low-level details, we usually define *cooperative subtasks* at high level(s) of the hierarchy. The proposed hierarchical approach allows agents to learn coordination faster by sharing information at the level of *cooperative subtasks*, rather than attempting to learn coordination at the level of primitive actions. We initially assume that communication is free and propose a hierarchical multi-agent RL algorithm called *Cooperative HRL*. In Section 6.4, we use a large four-agent AGV scheduling problem as the experimental testbed and compare the performance of the *Cooperative HRL* algorithm with selfish HRL, as well as single-agent HRL and standard Q-learning algorithms. We also show that the *Cooperative HRL* outperforms widely used industrial heuristics, such as *"first come first serve"*, *"highest queue first"* and *"nearest station first"* in this problem.

Later in this chapter, we address the issue of rational communication among autonomous agents, which is important when communication is costly. The goal is for agents to learn both action and communication policies that together optimize the task given the communication cost. We extend the *Cooperative HRL* algorithm to include communication decisions and propose a cooperative multi-agent HRL algorithm called *COM-Cooperative HRL*. In this algorithm, we add a communication level to the hierarchical decomposition of the problem below each *cooperation level*. Before making a decision at a *cooperative subtask*, agents decide if it is worthwhile to perform a communication action. A communication action has a certain cost and provides each agent at a certain *cooperation level* with the actions selected by the other agents at the same level. We demonstrate the efficacy of the *COM-Cooperative HRL* algorithm as well as the relation between the communication cost and the learned communication policy using a multi-agent taxi problem.

The rest of this chapter is organized as follows. In Section 6.1, we introduce the multi-agent SMDP model, which is an extension of the SMDP model to cooperative multi-agent domains. Section 6.2 describes the hierarchical multi-agent RL framework which is used in the algorithms proposed in this chapter. In Sections 6.3 and 6.4, we introduce the *Cooperative HRL* algorithm and present the experimental results of using this algorithm in a four-agent AGV scheduling problem. In Section 6.5, we illustrate how to incorporate communication decisions in the *Cooperative HRL* algorithm. In this section, after a brief introduction of communication among agents in Section 6.5.1, we illustrate the *COM-Cooperative HRL* algorithm in Section 6.5.2. Section 6.6 presents experimental results of using the *COM-Cooperative HRL* algorithm in a multi-agent taxi domain. Finally, Section 6.7 summarizes the chapter and discusses some directions for future work. The multi-agent version of the robot trash collection task described in Chapter 3 will serve as our example domain throughout this chapter. The multi-agent trash collection task and its task graph are shown in Figure 6.1.

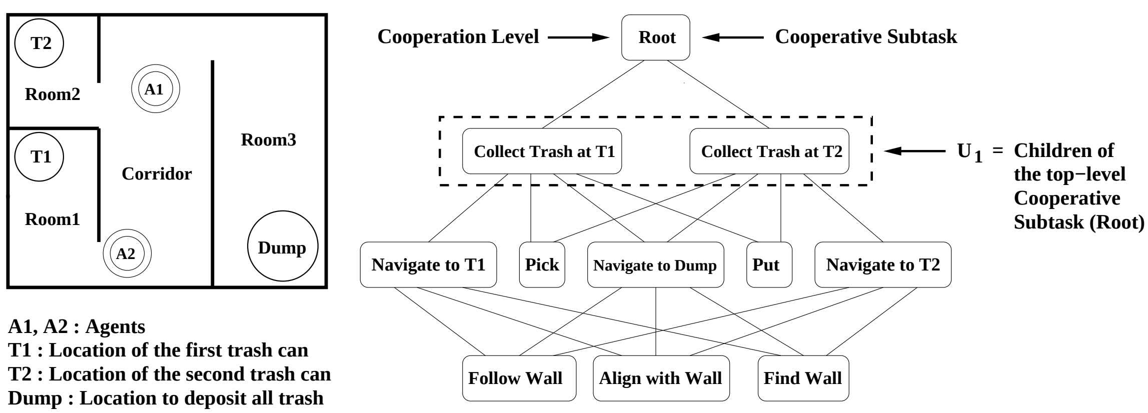


Figure 6.1. A multi-agent trash collection task and its associated task graph.

6.1 Multi-Agent SMDP Model

In this section, we extend the SMDP model described in Section 2.3 to multi-agent domains when a team of agents controls the process, and introduce the **multi-agent SMDP**

(MSMDP) model. We assume agents are cooperative, i.e., maximize the same utility over an extended period of time. The individual actions of agents interact in that the effect of one agent's action may depend on the actions taken by the others. When a group of agents perform temporally extended actions, these actions may not terminate at the same time. Therefore, unlike the multi-agent extension of an MDP, the MMDP model (Boutilier, 1999), the multi-agent extension of SMDP requires extending the notion of a decision making event.

Definition 6.1: An MSMDP consists of six components $(\Upsilon, \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, I, \mathcal{T})$, which are defined as follows:

The set Υ is a finite collection of n agents, with each agent $j \in \Upsilon$ having a finite set A^j of individual actions. An element $\vec{a} = \langle a^1, \ldots, a^n \rangle$ of the joint-action space $\mathcal{A} = \prod_{j=1}^{n} A^j$ represents the concurrent execution of actions a^j by each agent j, $j = 1, \ldots, n$. The components $\mathcal{S}$, $\mathcal{R}$, I, and $\mathcal{P}$ are as in an SMDP, the set of states of the system being controlled, the reward function mapping $\mathcal{S} \rightarrow \mathbb{R}$, the initial state distribution $I : \mathcal{S} \rightarrow [0, 1]$, and the state and action dependent multi-step transition probability function $\mathcal{P} : \mathcal{S} \times \mathbb{N} \times \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$. The term $P(s', N|s, \vec{a})$ denotes the probability that joint-action $\vec{a}$ will cause the system to transition from state s to state s' in N time steps. Since the components of a joint-action are temporally extended actions, they may not terminate at the same time. Therefore, the multi-step transition probability P depends on how we define decision epochs and as a result, depends on the termination scheme $\mathcal{T}$. Three termination strategies τ_{any}, τ_{all}, and $\tau_{continue}$ for temporally extended joint-actions were introduced and analyzed in Rohanimanesh and Mahadevan (2003). In τ_{any} termination scheme, the next decision epoch is when the first action within the joint-action currently being executed terminates, where the rest of the actions that did not terminate are interrupted. When an agent completes an action (e.g., finishes *collect trash at* $T1$ by putting trash in *Dump*), all other agents inter-

rupt their actions, the next decision epoch occurs, and a new joint-action is selected (e.g., agent $A1$ chooses to collect trash at $T2$ and agent $A2$ decides to collect trash at $T1$). In τ_{all} termination scheme, the next decision epoch is the earliest time at which all the actions within the joint-action currently being executed have terminated. When an agent completes an action, it waits (takes the *idle* action) until all the other agents finish their current actions. Then, next decision epoch occurs and agents choose next joint-action together. In both these termination strategies, all agents make decision at every decision epoch. The $\tau_{continue}$ termination scheme is similar to τ_{any} in the sense that the next decision epoch is when the first action within the joint-action currently being executed terminates. However, the other agents are not interrupted and only terminated agents select new actions. In this termination strategy, only a subset of agents choose action at each decision epoch. When an agent completes an action, next decision epoch occurs only for that agent and it selects its next action given the actions being performed by the other agents. □

The three termination strategies described above are the most common, but not the only termination schemes in cooperative multi-agent activities. A wide range of termination strategies can be defined based on them. Of course, not all these strategies are appropriate for any given multi-agent task. We categorize termination strategies as synchronous and asynchronous. In **synchronous** schemes, such as τ_{any} and τ_{all}, all agents make a decision at every decision epoch and therefore we need a centralized mechanism to synchronize agents at decision epochs. In **asynchronous** strategies, such as $\tau_{continue}$, only a subset of agents make decision at each decision epoch. In this case, there is no need for a centralized mechanism to synchronize agents and decision making can take place in a decentralized fashion. Since our goal is to design decentralized multi-agent RL algorithms, we use the $\tau_{continue}$ termination scheme for joint-action selection in the hierarchical multi-agent model and algorithms presented in this chapter.

6.2 A Hierarchical Multi-Agent Reinforcement Learning Framework

In our hierarchical multi-agent framework, we assume that there are n agents in the environment, cooperating with each other to accomplish a task. The designer of the system uses her/his domain knowledge to recursively decompose the overall task into a collection of subtasks that she/he believes are important for solving the problem. We assume that agents are **homogeneous**, i.e., all agents are given the same task hierarchy.[2] At each level of the hierarchy, the designer of the system defines **cooperative subtasks** to be those subtasks in which coordination among agents significantly increases the performance of the overall task. The set of all *cooperative subtasks* at a certain level of the hierarchy is called the **cooperation set** of that level. Each level of the hierarchy with non-empty *cooperation set* is called a **cooperation level**. The union of the children of the lth level *cooperative subtasks* is represented by U_l. Since high-level coordination allows for increased cooperation skills as agents do not get confused by low-level details, we usually define *cooperative subtasks* at the highest level(s) of the hierarchy. Agents actively coordinate while making decision in *cooperative subtasks* and are ignorant about other agents in **non-cooperative subtasks**. Thus, we configure *cooperative subtasks* to model joint-action values. In the trash collection problem, we define *root* as a *cooperative subtask*. As a result, the top-level of the hierarchy is a *cooperation level*, *root* is the only member of the *cooperation set* at the top-level, and U_1 consists of all subtasks located at the second level of the hierarchy, $U_1 = \{collect\ trash\ at\ T1, collect\ trash\ at\ T2\}$ (see Figure 6.1). As it is clear in this problem, it is more efficient that an agent learns high-level coordination knowledge (what is the utility of agent $A2$ collecting trash from trash can $T1$ if agent $A1$ is collecting trash from trash can $T2$), rather than learning its response to low-level primitive actions of other agents (what agent $A2$ should do if agent $A1$ aligns with wall). Therefore, we define single-agent policies for *non-cooperative subtasks* and joint policies for *cooperative subtasks*.

[2] Studying the **heterogeneous** case where agents are given dissimilar decompositions of the overall task would be more challenging and beyond the scope of this dissertation.

Definition 6.2: Under a hierarchical policy $\boldsymbol{\mu}$, each *non-cooperative subtask* M_i can be modeled by an SMDP consisting of components $(S_i, A_i, P_i^{\boldsymbol{\mu}}, R_i)$. $\square$

Definition 6.3: Under a hierarchical policy $\boldsymbol{\mu}$, each *cooperative subtask* M_i located at the lth level of the hierarchy can be modeled by an MSMDP as follows:

Υ is the set of n agents in the team. We assume that agents have only local state information and ignore the states of the other agents. Therefore, the state space $\mathcal{S}_i$ is defined as the single-agent state space S_i (not joint-state space). This is certainly an approximation but greatly simplifies the underlying multi-agent RL problem. This approximation is based on the fact that an agent can get a rough idea of what state the other agents might be in just by knowing the high-level actions being performed by them. The action space is joint and is defined as $\mathcal{A}_i = A_i \times (U_l)^{n-1}$, where $U_l = \bigcup_{k=1}^{m} A_k$ is the union of the action sets of all the lth level *cooperative subtasks*, and m is the cardinality of the lth level *cooperation set*. For the cooperative subtask *root* in the trash collection problem, the set of agents is $\Upsilon = \{A1, A2\}$ and its joint-action space, $\mathcal{A}_{root}$, is specified as the cross product of its action set, A_{root}, and U_1, $\mathcal{A}_{root} = A_{root} \times U_1$. Finally, since we are interested in decentralized control, we use the $\tau_{continue}$ termination strategy. Therefore, when an agent terminates a subtask, the next decision epoch occurs only for that agent and it selects its next action given the information about the other agents. $\square$

This cooperative multi-agent approach has the following pros and cons:

Pros

- Using HRL scales learning to problems with large state spaces by using the task structure to restrict the space of policies.

- Cooperation among agents is faster and more efficient as agents learn joint-action values only at *cooperative subtasks* usually located at the high level(s) of abstraction and do not get confused by low-level details.

- Since high-level subtasks can take a long time to complete, communication is needed only fairly infrequently.

- The complexity of the problem is reduced by storing only the local state information by each agent. It is due to the fact that each agent can often get a rough idea of the state of the other agents just by knowing about their high-level actions.

Cons

- The learned policy would not be optimal if agents need to coordinate at the subtasks that have not been defined as *cooperative*. This issue will be addressed in one of the AGV experiments in Section 6.4, by extending the joint-action model to the lower levels of the hierarchy. Although this extension provides the cooperation required at the lower levels, it increases the number of parameters to be learned and as a result the complexity of the learning problem.

- If communication is costly, this method might not find an appropriate policy for the problem. We address this issue in Section 6.5 by including communication decisions in the model. If communication is cheap, agents learn to cooperate with each other, and if communication is expensive, agents prefer to make decision only based on their local view of the overall problem.

- Storing only local state information by agents causes sub-optimality in general. On the other hand, including the state of the other agents dramatically increases the complexity of the learning problem and has its own inefficacy. We do not explicitly address this problem in this dissertation.

The value function decomposition described in Section 3.5 relies on a key principle: the reward function for the parent task is the value function of the child task (see Equations 3.4 and 3.5). Now, we show how the single-agent two-part value function decomposition described in Section 3.5 can be modified to formulate the joint-value function for *cooperative subtasks*. In our hierarchical multi-agent model, we configure *cooperative subtasks* to store the **joint completion function** values.

Definition 6.4: The joint completion function for agent j, $C^j(i, s, a^1, \dots, a^{j-1}, a^{j+1}, \dots, a^n, a^j)$, is the expected discounted cumulative reward of completing *cooperative subtask* M_i after taking subtask a^j in state s while other agents performing subtasks $a^k, \forall k \in \{1, \dots, n\}, k \neq j$. The reward is discounted back to the point in time where a^j begins execution. □

In this definition, M_i is a *cooperative subtask* at level l of the hierarchy and $\langle a^1, \dots, a^n \rangle$ is a joint-action in the action set of M_i. Each individual action in this joint-action belongs to U_l. More precisely, the decomposition equations used for calculating the projected value and action-value function for *cooperative subtask* M_i of agent j have the following form:

$$\hat{V}^j(i, s, a^1, \dots, a^{j-1}, a^{j+1}, \dots, a^n) = \hat{Q}^j(i, s, a^1, \dots, a^{j-1}, a^{j+1}, \dots, a^n, \mu_i^j(s))$$

$$\hat{Q}^j(i, s, a^1, \dots, a^{j-1}, a^{j+1}, \dots, a^n, a^j) = \hat{V}^j(a^j, s) + C^j(i, s, a^1, \dots, a^{j-1}, a^{j+1}, \dots, a^n, a^j) \tag{6.1}$$

One important point to note in this equation is that if subtask a^j is itself a *cooperative subtask* at level $l+1$ of the hierarchy, its projected value function is defined as a joint projected value function $\hat{V}^j(a^j, s, \tilde{a}^1, \dots, \tilde{a}^{j-1}, \tilde{a}^{j+1}, \dots, \tilde{a}^n)$, where $\tilde{a}^1, \dots, \tilde{a}^{j-1}, \tilde{a}^{j+1}, \dots, \tilde{a}^n$ belong to U_{l+1}. In this case, in order to calculate $\hat{V}^j(a^j, s)$ for Equation 6.1, we marginalize $\hat{V}^j(a^j, s, \tilde{a}^1, \dots, \tilde{a}^{j-1}, \tilde{a}^{j+1}, \dots, \tilde{a}^n)$ over $\tilde{a}^1, \dots, \tilde{a}^{j-1}, \tilde{a}^{j+1}, \dots, \tilde{a}^n$.

We illustrate the above projected joint-value function decomposition using the trash collection task. The two-part value function decomposition for agent $A1$ at *root* has the following form:

$$\begin{aligned}\hat{Q}^1(root, s, collect\ trash\ at\ T\,2\,, collect\ trash\ at\ T1) &= \hat{V}^1(collect\ trash\ at\ T1, s)\\ &+C^1(root, s, collect\ trash\ at\ T2, collect\ trash\ at\ T1)\end{aligned}$$

which represents the value of agent $A1$ performing *collect trash at T1* in the context of the overall task (*root*), when agent $A2$ is executing *collect trash at T2*. Note that this value is decomposed into the projected value of *collect trash at T1* subtask (the $\hat{V}$ term), and the completion value of the remainder of the *root* task (the C term).

Given a hierarchical decomposition for any problem, we need to find the highest level subtasks at which decomposition Equation 6.1 provides a sufficiently good approximation of the true value. For the problems used in the experiments of this chapter, coordination only at the highest level of the hierarchy is a good compromise between achieving a desirable performance and reducing the number of joint-state-action values that need to be learned. Hence, we define *root* as a *cooperative subtask* and thus the highest level of the hierarchy as a *cooperation level* in these experiments. We extend coordination to lower levels of the hierarchy by defining *cooperative subtasks* at levels below *root* in one of the experiments of Section 6.4.

6.3 A Hierarchical Multi-Agent Reinforcement Learning Algorithm

In this section, we use the hierarchical multi-agent RL framework described in Section 6.2 and present a hierarchical multi-agent RL algorithm, called **Cooperative HRL**. The pseudo code for this algorithm is shown in Algorithm 4 at the end of this chapter. In the *Cooperative HRL*, $\hat{V}$ and C values can be learned through a standard TD-learning method based on sample trajectories. One important point to note is that since non-primitive subtasks are temporally extended in time, the update rules for C values used in this algorithm

are based on the SMDP model. In this algorithm, an agent starts from the *root* task and chooses a subtask till it reaches a primitive action i. It executes primitive action i in state s, receives reward r and observes resulting state s', the value function V of primitive subtask[3] M_i is updated using:

$$V_{t+1}(i, s) = [1 - \alpha_t(i)]V_t(i, s) + \alpha_t(i)r$$

where $\alpha_t(i)$ is the learning rate for subtask M_i at time t. This parameter should be gradually decreased to zero in time limit.

Whenever a subtask terminates, the C values are updated for all states visited during the execution of that subtask. Assume an agent is executing a non-primitive subtask M_i and is in state s, then while subtask M_i does not terminate, it chooses subtask M_a according to the current exploration policy (softmax or ϵ-greedy with respect to $\mu_i(s)$). If subtask M_a takes N primitive steps and terminates in state s', the corresponding C value is updated using

$$C_{t+1}(i, s, a) = [1 - \alpha_t(i)]C_t(i, s, a) + \alpha_t(i)\gamma^N[C_t(i, s', a^*) + \hat{V}_t(a^*, s')] \tag{6.2}$$

where $a^* = \arg\max_{a' \in A_i}[C_t(i, s', a') + \hat{V}_t(a', s')]$.

The $\hat{V}$ values in Equation 6.2 are calculated using the following equation:

$$\hat{V}(i, s) = \begin{cases} \max_{a \in A_i} \hat{Q}(i, s, a) & \text{if } M_i \text{ is a non-primitive subtask,} \\ \sum_{s' \in S_i} P(s'|s, i)r(s, i) & \text{if } M_i \text{ is a primitive action.} \end{cases}$$

Similarly, when agent j completes execution of subtask $a^j \in A_i$, the joint completion function C of *cooperative subtask* M_i located at level l of the hierarchy is updated for all the states visited during the execution of subtask a^j using

[3]We do not use $\hat{V}$ here, since projected and hierarchical value functions are the same for primitive actions.

$$C_{t+1}^{j}(i,s,a^{1},\dots,a^{j-1},a^{j+1},\dots,a^{n},a^{j}) = [1-\alpha_{t}^{j}(i)]C_{t}^{j}(i,s,a^{1},\dots,a^{j-1},a^{j+1},\dots,a^{n},a^{j})$$
$$+\alpha_{t}^{j}(i)\gamma^{N}[C_{t}^{j}(i,s',\hat{a}^{1},\dots,\hat{a}^{j-1},\hat{a}^{j+1},\dots,\hat{a}^{n},a^{*})+\hat{V}_{t}^{j}(a^{*},s')] \tag{6.3}$$

where $a^{*} = \arg\max_{a'\in A_i}[C_{t}^{j}(i,s',\hat{a}^{1},\dots,\hat{a}^{j-1},\hat{a}^{j+1},\dots,\hat{a}^{n},a')+\hat{V}_{t}^{j}(a',s')]$, $a^{1},\dots,a^{j-1},a^{j+1},\dots,a^{n}$ and $\hat{a}^{1},\dots,\hat{a}^{j-1},\hat{a}^{j+1},\dots,\hat{a}^{n}$ are actions in U_l being performed by the other agents when agent j is in states s and s' respectively.

Equation 6.3 indicates that in addition to the states visited during the execution of a subtask in U_l (s and s'), an agent must store the actions in U_l being performed by all the other agents ($a^{1},\dots,a^{j-1},a^{j+1},\dots,a^{n}$ in state s and $\hat{a}^{1},\dots,\hat{a}^{j-1},\hat{a}^{j+1},\dots,\hat{a}^{n}$ in state s'). Sequence *Seq* is used for this purpose in Algorithm 4.

6.4 Experimental Results for the Cooperative HRL Algorithm

In this section, we demonstrate the performance of the *Cooperative HRL* algorithm proposed in Section 6.3 using a four-agent AGV scheduling task. In this experiment, we first provide a brief overview of the domain, then apply the *Cooperative HRL* algorithm to the problem, and finally compare its performance with other algorithms, such as selfish multi-agent HRL (where each agent acts independently and learns its own optimal policy), single-agent HRL, and flat Q-Learning.

Figure 6.2 shows the layout of the AGV scheduling domain. $M1$ to $M4$ show workstations in this environment. Parts of type i have to be carried to the drop-off station at workstation i, D_i, and the assembled parts brought back from pick-up stations of workstations, P_i's, to the warehouse. The AGV travel is unidirectional (as the arrows show). This task is decomposed using the task graph in Figure 6.3. Each agent uses a copy of this task graph. We define *root* as a *cooperative subtask* and the highest level of the hierarchy as a *cooperation level*. Therefore, all subtasks at the second level of the hierarchy

Algorithm 4 The Cooperative HRL algorithm.

Function Cooperative-HRL(Agent j, Task M_i at the lth level of the hierarchy, State s)
let *Seq* = {} be the sequence of (*state-visited, actions in* $\bigcup_{k=1}^{L} U_k$ *being performed by the other agents*) while executing M_i /* L is the number of levels in the hierarchy */
if M_i is a primitive action **then**
  execute action i in state s, receive reward $r(s,i)$ and observe state s'
  $V_{t+1}^{j}(i,s) \longleftarrow [1-\alpha_t^j(i)]V_t^j(i,s) + \alpha_t^j(i) r(s,i)$
  push (*state* s, *actions in* $\{U_l | l\ is\ a\ cooperation\ level\}$ *being performed by the other agents*) onto the front of *Seq*
else /* M_i is a non-primitive subtask */
  while M_i has not terminated **do**
    if M_i is a *cooperative subtask* **then**
      choose action a^j according to the current exploration policy $\mu_i^j(s, a^1, \ldots, a^{j-1}, a^{j+1}, \ldots, a^n)$
      let *ChildSeq* = Cooperative-HRL(M_j, a^j, s), where *ChildSeq* is the sequence of (*state-visited, actions in* $\bigcup_{k=1}^{L} U_k$ *being performed by the other agents*) while executing action a^j
      observe result state s' and $\hat{a}^1, \ldots, \hat{a}^{j-1}, \hat{a}^{j+1}, \ldots, \hat{a}^n$ actions in U_l being performed by the other agents
      let $a^* = \arg\max_{a' \in A_i}[C_t^j(i, s', \hat{a}^1, \ldots, \hat{a}^{j-1}, \hat{a}^{j+1}, \ldots, \hat{a}^n, a') + \hat{V}_t^j(a', s')]$
      let $N = 0$
      for each (s, $a^1, \ldots, a^{j-1}, a^{j+1}, \ldots, a^n$) in *ChildSeq* from the beginning **do**
        $N = N + 1$
        $C_{t+1}^j(i, s, a^1, \ldots, a^{j-1}, a^{j+1}, \ldots, a^n, a^j) \longleftarrow$
        $[1-\alpha_t^j(i)]C_t^j(i, s, a^1, \ldots, a^{j-1}, a^{j+1}, \ldots, a^n, a^j) +$
        $\alpha_t^j(i)\gamma^N[C_t^j(i, s', \hat{a}^1, \ldots, \hat{a}^{j-1}, \hat{a}^{j+1}, \ldots, \hat{a}^n, a^*) + \hat{V}_t^j(a^*, s')]$
      end for
    else /* M_i is not a *cooperative subtask* */
      choose action a^j according to the current exploration policy $\mu_i^j(s)$
      let *ChildSeq* = Cooperative-HRL(M_j, a^j, s), where *ChildSeq* is the sequence of (*state-visited, actions in* $\bigcup_{k=1}^{L} U_k$ *being performed by the other agents*) while executing action a^j
      observe result state s'
      let $a^* = \arg\max_{a' \in A_i}[C_t^j(i, s', a') + \hat{V}_t^j(a', s')]$
      let $N = 0$
      for each state s in *ChildSeq* from the beginning **do**
        $N = N + 1$
        $C_{t+1}^j(i, s, a^j) \longleftarrow [1-\alpha_t^j(i)]C_t^j(i, s, a^j) + \alpha_t^j(i)\gamma^N[C_t^j(i, s', a^*) + \hat{V}_t^j(a^*, s')]$
      end for
    end if
    append *ChildSeq* onto the front of *Seq*
    $s = s'$
  end while
end if
return *Seq*
end Cooperative-HRL

($DM1, \ldots, DM4, DA1, \ldots, DA4$) belong to set U_1. Coordination skills among agents are learned by using joint-action values at the highest level of the hierarchy as described in Section 6.3.

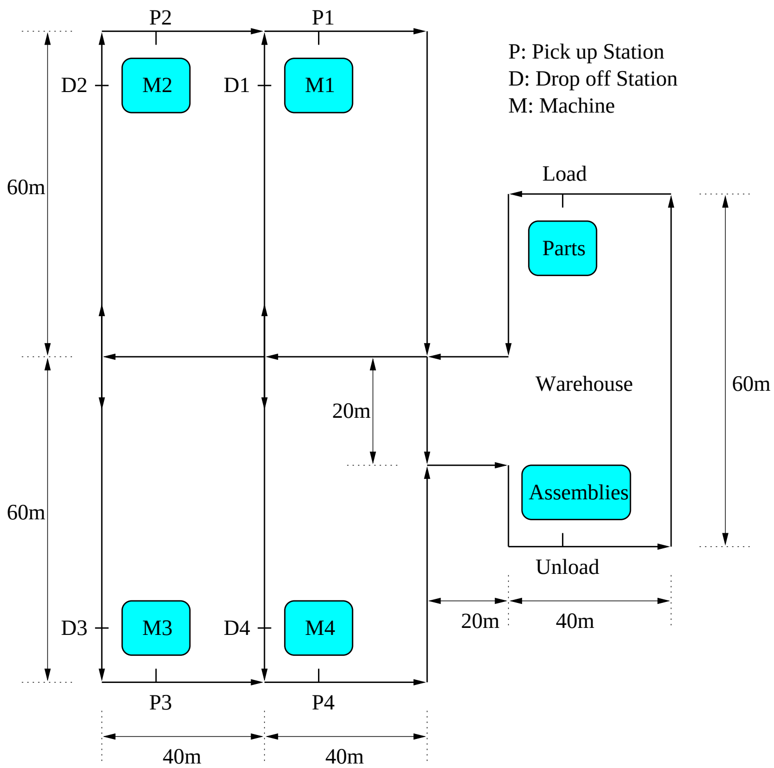


Figure 6.2. A multi-agent AGV scheduling domain. There are four AGVs (not shown) which carry raw materials and finished parts between machines and the warehouse.

The state of the environment consists of the number of parts in the pick-up and drop-off stations of each machine, and whether the warehouse contains parts of each of the four types. In addition, each agent keeps track of its own location and status as a part of its state space. Thus, in the flat case, the state space consists of 100 locations, 8 buffers of size 3, 9 possible states of AGV (carrying part1, $\ldots$, carrying assembly1, $\ldots$, empty), and 2 values for each part in the warehouse, i.e., $100 \times 4^8 \times 9 \times 2^4 \approx 10^9$ states. The state abstraction helps in reducing the state space considerably. Only the relevant state variables are used while storing the completion functions in each node of the task graph. For example, for the navigation subtasks, only the *location* state variable is relevant, and this subtask can

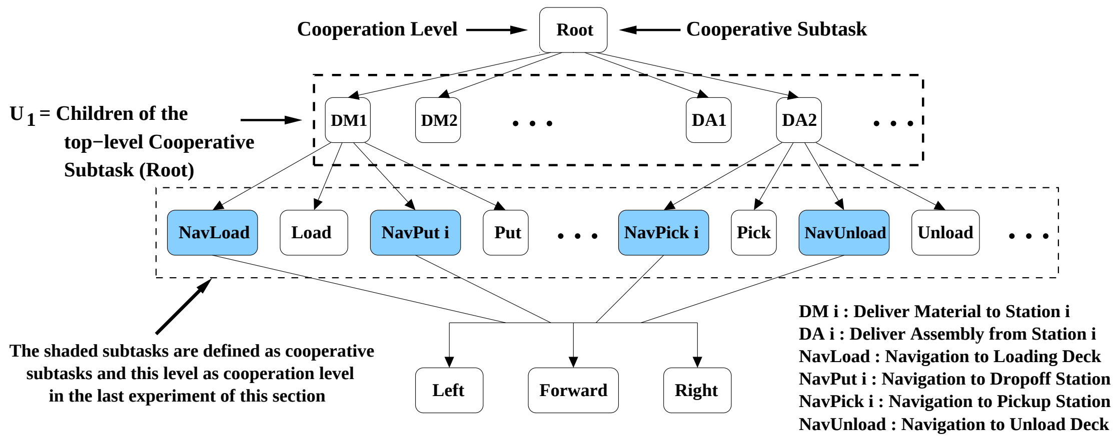


Figure 6.3. Task graph for the AGV scheduling task.

be learned with 100 values. Hence, for each high-level subtask ($DM1, \ldots, DM4$), the number of relevant states would be $100 \times 9 \times 4 \times 2 = 7,200$, and for each high-level subtask ($DA1, \ldots, DA4$), the number of relevant states would be $100 \times 9 \times 4 = 3,600$. This state abstraction gives us a compact way of representing the C and V functions, and speeds up the algorithm.

In the experiments of this section, we assume that there are four agents (AGVs) in the environment. The experimental results were generated with the following model parameters. The inter-arrival time for parts at the warehouse is uniformly distributed with a mean of 4 sec and variance of 1 sec. The percentage of *Part1*, *Part2*, *Part3*, and *Part4* in the part arrival process are 20, 28, 22, and 30 respectively. The time required for assembling the various parts is normally distributed with means 15, 24, 24, and 30 sec for *Part1*, *Part2*, *Part3*, and *Part4* respectively, and variance 2 sec. The execution time of primitive actions (*right*, *left*, *forward*, *load*, and *unload*) is normally distributed with mean 1000 μ-sec and variance 50 μ-sec. The execution time for the *idle* action is also normally distributed with mean 1 sec and variance 0.1 sec. Table 6.1 summarizes the values of the model parameters used in the experiments of this section. In this task, each experiment was conducted five times and the results were averaged.

Parameter	Distribution	Mean (sec)	Variance (sec)
Idle Action	Normal	1	0.1
Primitive Actions	Normal	0.001	0.00005
Assembly Time for Part1	Normal	15	2
Assembly Time for Part2	Normal	24	2
Assembly Time for Part3	Normal	24	2
Assembly Time for Part4	Normal	30	2
Inter-Arrival Time for Parts	Uniform	4	1

Table 6.1. Model parameters for the multi-agent AGV scheduling task.

Figure 6.4 shows the throughput of the system for the three algorithms, single-agent HRL, selfish multi-agent HRL, and *Cooperative HRL*. As seen in Figure 6.4, agents learn a little faster initially in the selfish multi-agent method, but after some time the algorithm results in sub-optimal performance. This is due to the fact that two or more agents select the same action, but once the first agent completes the task, the other agents might have to wait for a long time to complete the task, due to the constraints on the number of parts that can be stored at a particular place. The system throughput achieved using the *Cooperative HRL* method is higher than the single-agent HRL and the selfish multi-agent HRL algorithms. This difference is even more significant in Figure 6.5, when the primitive actions have longer execution time, almost $\frac{1}{10^{th}}$ of the average assembly time (the mean execution time of primitive actions is 2 sec).

Figure 6.6 shows the results from an implementation of the single-agent flat Q-Learning with the buffer capacity at each station set at 1. As can be seen from the plot, the flat algorithm converges extremely slowly. The throughput at $70,000$ sec has gone up to only 0.07, compared with 2.6 for the hierarchical single-agent case. Figure 6.7 compares the *Cooperative HRL* algorithm with several well-known AGV scheduling rules, *highest queue first*, *nearest station first*, and *first come first serve*, showing clearly the improved performance of the HRL method.

So far in our experiments in the AGV domain, we only defined *root* as a *cooperative subtask*. Now in our last experiment in this domain, in addition to *root*, we define navi-

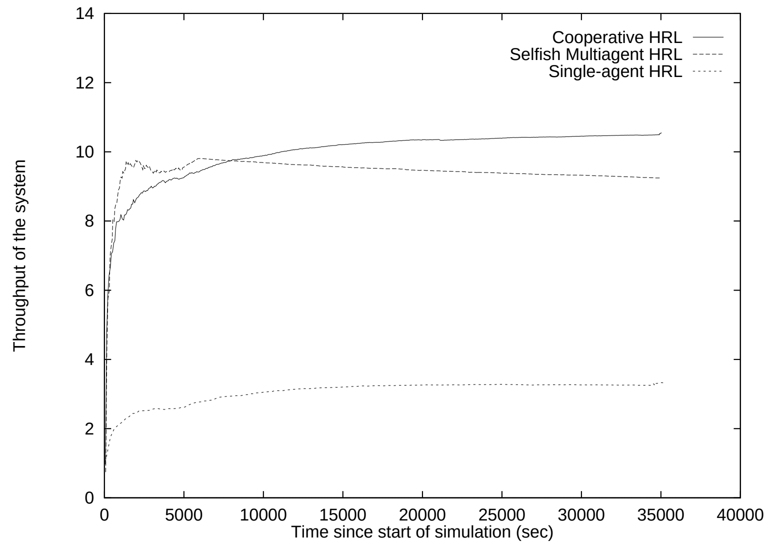


Figure 6.4. This figure shows that the *Cooperative HRL* algorithm outperforms both the selfish multi-agent HRL and the single-agent HRL algorithms when the AGV travel time and load/unload time are very much less compared to the average assembly time.

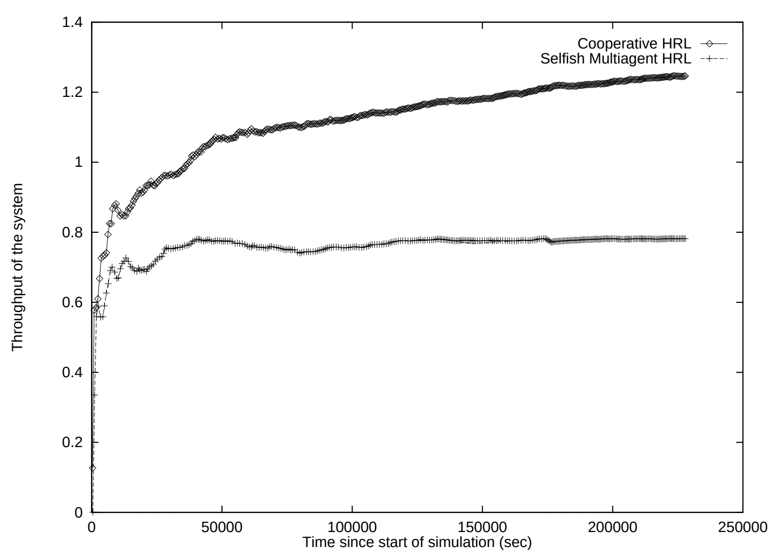


Figure 6.5. This figure compares the *Cooperative HRL* algorithm with the selfish multi-agent HRL, when the AGV travel time and load/unload time are $\frac{1}{10^{th}}$ of the average assembly time.

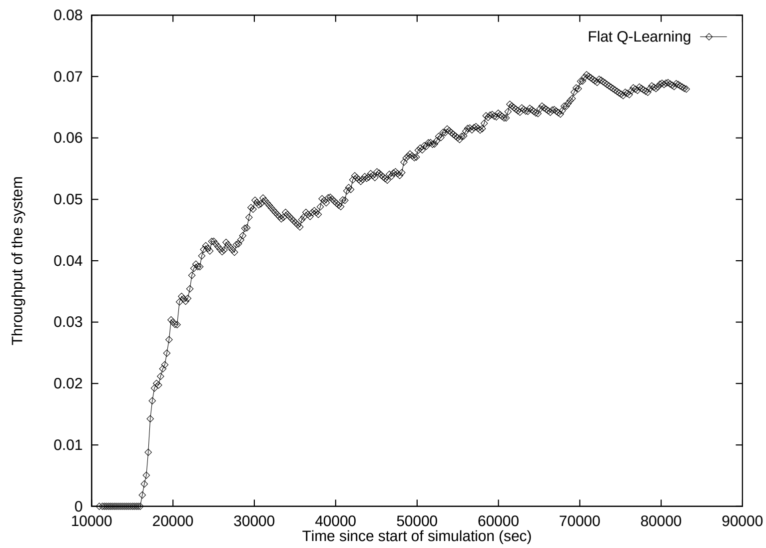


Figure 6.6. A flat Q-Learner learns the AGV domain extremely slowly showing the need for using a hierarchical task structure.

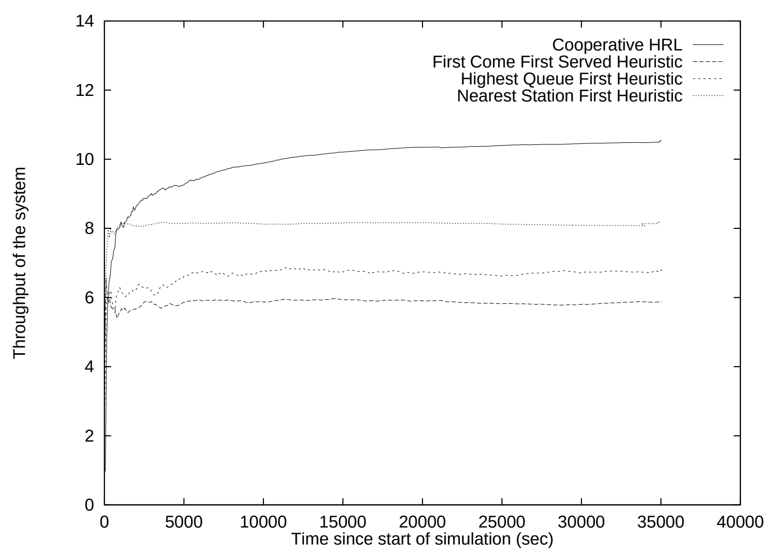


Figure 6.7. This plot shows that the *Cooperative HRL* algorithm outperforms three well-known widely used industrial heuristics for AGV scheduling.

gation subtasks at the third level of the hierarchy as *cooperative subtasks*. Therefore, the third level of the hierarchy is also a *cooperation level* and its *cooperation set* contains all navigation subtasks at that level (see Figure 6.3). We configure the *root* and the third level navigation subtasks to represent joint-actions. Figure 6.8 compares the performance of the system in these two cases. When the navigation subtasks are configured to represent joint-actions, learning is considerably slower (since the number of parameters is increased significantly) and the overall performance is not better. The lack of improvement is due in part to the fact that the AGV travel is unidirectional, as shown in Figure 6.2, thus coordination at the navigation level does not improve the performance of the system. However, there exist problems that adding joint-actions in multiple levels will be worthwhile, even if convergence is slower, due to better overall performance.

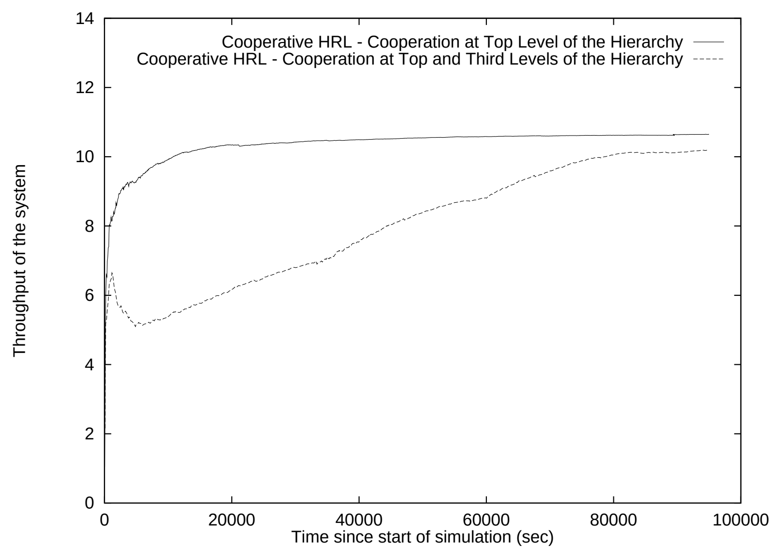


Figure 6.8. This plot compares the performance of the *Cooperative HRL* algorithm with cooperation at the top level of the hierarchy vs. cooperation at the top and third levels of the hierarchy.

6.5 Hierarchical Multi-Agent RL with Communication Decisions

Communication is used by agents to obtain local information of their teammates by paying a certain cost. The *Cooperative HRL* algorithm described in Section 6.3 works under three important assumptions, free, reliable, and instantaneous communication, i.e., communication cost is zero, no message is lost in the environment, and each agent has enough time to receive information about its teammates before taking its next action. Since communication is free, as soon as an agent selects an action at a *cooperative subtask*, it broadcasts it to the team. Using this simple rule, and the fact that communication is reliable and instantaneous, whenever an agent is about to choose an action at an lth level *cooperative subtask*, it knows the subtasks in U_l being performed by all its teammates.

However, communication can be costly and unreliable in real-world problems. When communication is not free, it is no longer optimal for a team that agents always broadcast actions taken at their *cooperative subtasks* to their teammates. Therefore, agents must learn to optimally use communication by taking into account its long term return and its immediate cost. In the remainder of this chapter, we examine the case that communication is not free, but still assume that it is reliable and instantaneous. In this section, we first describe the communication framework and then illustrate how we extend the *Cooperative HRL* algorithm to include communication decisions and propose a new algorithm, called **COM-Cooperative HRL**. The goal of this algorithm is to learn a hierarchical policy (a set of policies for all subtasks including the communication subtasks) to maximize the team utility given the communication cost. Finally, in Section 6.6, we demonstrate the efficacy of the *COM-Cooperative HRL* algorithm as well as the relation between the communication cost and the learned communication policy using a multi-agent taxi domain.

6.5.1 Communication Among Agents

Communication usually consists of three steps: *send*, *answer*, and *receive*. At the **send** step t_s, agent j decides if communication is necessary, performs a communication ac-

tion, and sends a message to agent i. At the **answer** step $t_a \geq t_s$, agent i receives the message from agent j, updates its local information using the contents of the message (if necessary), and sends back the answer (if required). At the **receive** step $t_r \geq t_a$, agent j receives the answer of its message, updates its local information, and decides on which non-communicative action to execute. Generally there are two types of messages in a communication framework: **request** and **inform**. For simplicity, we suppose that relative ordering of messages do not change, which means that for two communication actions c_1 and c_2, if $t_s(c_1) < t_s(c_2)$ then $t_a(c_1) \leq t_a(c_2)$ and $t_r(c_1) \leq t_r(c_2)$. The following three types of communication actions are commonly used in a communication model:

- $Tell(j, i)$: agent j sends an *inform* message to agent i.
- $Ask(j, i)$: agent j sends a *request* message to agent i, which is answered by agent i with an *inform* message.
- $Sync(j, i)$: agent j sends an *inform* message to agent i, which is answered by agent i with an *inform* message.

In the *Cooperative HRL* algorithm described in Section 6.3, we assume free, reliable and instantaneous communication. Hence, the communication protocol of this algorithm is as follows: whenever an agent chooses an action at a *cooperative subtask*, it executes a *Tell* communication action and sends its selected action as an *inform* message to all other agents. As a result, when an agent is going to choose an action at an lth level *cooperative subtask*, it knows actions being performed by all other agents in U_l. *Tell* and *inform* are the only communication action and type of message used in the communication protocol of the *Cooperative HRL* algorithm.

6.5.2 A Hierarchical Multi-Agent RL Algorithm with Communication Decisions

When communication is costly in the *Cooperative HRL* algorithm, it is no longer optimal for the team that each agent broadcasts its action to all its teammates. In this case,

each agent must learn to optimally use the communication. To address the communication cost in the *COM-Cooperative HRL* algorithm, we add a communication level to the task graph of the problem below each *cooperation level*, as shown in Figure 6.9 for the trash collection task. In this algorithm, when an agent is going to make a decision at an lth level *cooperative subtask*, it first decides whether to communicate (takes **Communicate** action) with the other agents to acquire their actions in U_l, or do not communicate (takes **Not-Communicate** action) and selects its action without inquiring new information about its teammates. Agents decide about communication by comparing the expected value of communication plus the communication cost, $\hat{Q}(Parent(Com), s, Com) + ComCost$, with the expected value of not communicating with the other agents, $\hat{Q}(Parent(NotCom), s, Not-Com)$. If agent j decides not to communicate, it chooses an action like a selfish agent by using its action-value (not joint-action-value) function $\hat{Q}^j(NotCom, s, a)$, where $a \in Children(NotCom)$. When it decides to communicate, it first takes communication action $Ask(j, i), \forall i \in \{1, \ldots, j-1, j+1, \ldots, n\}$, where n is the number of agents, and sends a *request* message to all other agents. Other agents reply by taking communication action $Tell(i, j)$ and send their action in U_l as an *inform* message to agent j. Then agent j uses its joint-action-value (not action-value) function $\hat{Q}^j(Com, s, a^1, \ldots, a^{j-1}, a^{j+1}, \ldots, a^n, a)$, $a \in Children(Com)$ to select its next action in U_l. For instance, in the trash collection task, when agent $A1$ dumps trash and is going to move to one of the two trash cans, it should first decide whether to communicate with agent $A2$ in order to inquire its action in $U_1 = \{collect\ trash\ at\ T1,\ collect\ trash\ at\ T2\}$ or not. To make a communication decision, agent $A1$ compares $\hat{Q}^1(Root, s\ , NotCom)$ with $\hat{Q}^1(Root, s, Com) + ComCost$. If it chooses not to communicate, it selects its action using $\hat{Q}^1(NotCom, s, a)$, where $a \in U_1$. If it decides to communicate, after acquiring the action of agent $A2$ in U_1, a^{A2}, it selects its own action using $Q^1(Com, s, a^{A2}, a)$, where a and a^{A2} both belong to U_1.

In the *COM-Cooperative HRL*, we assume that when an agent decides to communicate, it communicates with all other agents as described above. We can make the model more

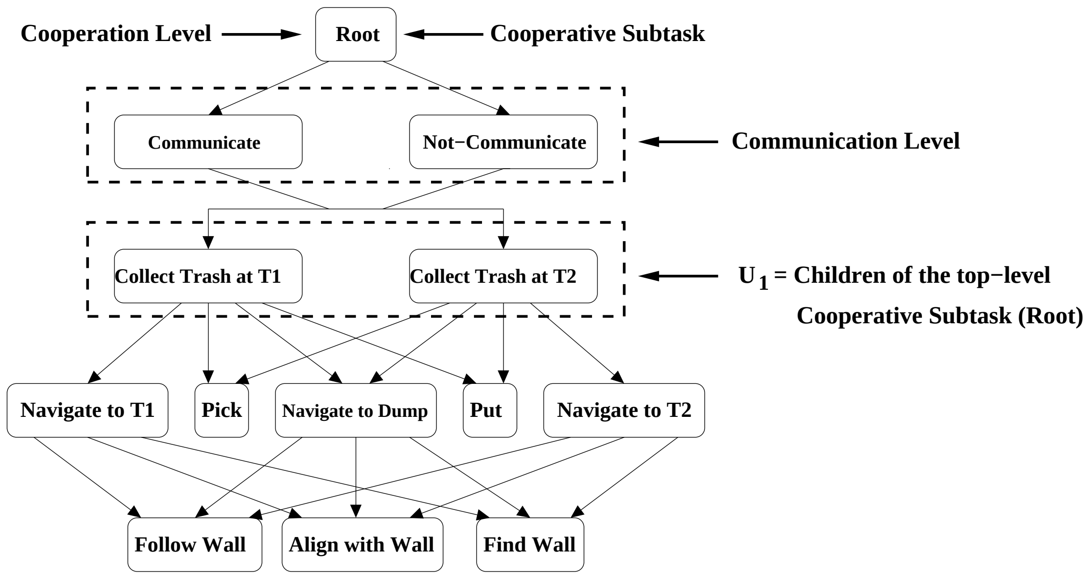


Figure 6.9. Task graph of the trash collection problem with communication actions.

complicated by making decision about communication with each individual agent. In this case, the number of communication actions would be $C^1_{n-1} + C^2_{n-1} + \ldots + C^{n-1}_{n-1}$, where C^q_p is the number of distinct combinations selecting q out of p agents. For instance, in a three-agent case, communication actions for agent 1 would be *communicate with agent 2*, *communicate with agent 3*, and *communicate with both agents 2 and 3*. It increases the number of communication actions and therefore the number of parameters to be learned. However, there are methods to reduce the number of communication actions in real-world applications. For instance, we can cluster agents based on their role in the team and assume each cluster as a single entity to communicate with. It reduces n from the number of agents to the number of clusters.

In the *COM-Cooperative HRL* algorithm, *Communicate* subtasks are configured to store joint completion function values, and *Not-Communicate* subtasks are configured to store completion function values. The joint completion function for agent j, $C^j(Com, s, a^1, \ldots,$ $a^{j-1}, a^{j+1}, \ldots, a^n, a^j)$ is defined as the expected discounted reward of completing subtask a^j by agent j in the context of the parent task Com, when other agents performing sub-

tasks $a^i, \forall i \in \{1, \ldots, j-1, j+1, \ldots, n\}$. In the trash collection domain, if agent $A1$ communicates with agent $A2$, its value function decomposition would be

$$\hat{Q}^1(Com, s, Collect\ Trash\ at\ T2, Collect\ Trash\ at\ T1) = \hat{V}^1(Collect\ Trash\ at\ T1, s) + C^1(Com, s, Collect\ Trash\ at\ T2, Collect\ Trash\ at\ T1)$$

which represents the projected value of agent $A1$ performing subtask *collect trash at T1,* when agent $A2$ is executing subtask *collect trash at T2*. Note that this value is decomposed into the projected value of subtask *collect trash at T1* and the value of completing subtask $Parent(Com)$ (here $root$ is the parent of subtask Com) after executing subtask *collect trash at T1*. If agent $A1$ does not communicate with agent $A2$, its value function decomposition would be

$$\begin{aligned}\hat{Q}^1(NotCom, s, Collect\ Trash\ at\ T1) = \hat{V}^1&(Collect\ Trash\ at\ T1, s)\\ &+ C^1(NotCom, s, Collect\ Trash\ at\ T1)\end{aligned}$$

which represents the projected value of agent $A1$ performing subtask *collect trash at T1,* regardless of the action being executed by agent $A2$.

Again, the $\hat{V}$ and C values are learned through a standard TD-learning method based on sample trajectories similar to the one presented in Algorithm 4. Completion function values for an action in U_l is updated when we take an action under *Not-Communicate* subtask, and joint completion function values for an action in U_l is updated when it is selected under *Communicate* subtask. In the later case, the actions selected in U_l by the other agents are known as a result of communication and are used to update the joint completion function values.

6.6 Experimental Results for the COM-Cooperative HRL Algorithm

In this section, we demonstrate the performance of the *COM-Cooperative HRL* algorithm proposed in Section 6.5.2 using a multi-agent taxi problem. We also investigate the relation between the communication policy and the communication cost in this domain.

Consider a 5-by-5 grid world inhabited by two taxis ($T1$ and $T2$) shown in Figure 6.10. There are four stations in this domain, marked as B(lue), G(reen), R(ed), and Y(ellow). The task is continuing, passengers appear according to a fixed passenger arrival rate[4] at these four stations and wish to be transported to one of the other stations chosen randomly. Taxis must go to the location of a passenger, pick up the passenger, go to her/his destination station, and drop the passenger there. The goal here is to increase the throughput of the system, which is measured in terms of the number of passengers dropped off at their destinations per $5,000$ time steps, and to reduce the average waiting time per passenger. This problem can be decomposed into subtasks and the resulting task graph is shown in Figure 6.10. Taxis need to learn three skills here. First, how to do each subtask, such as *navigate* to B, G, R, or Y, and when to perform *Pickup* or *Putdown* action. Second, the order to do the subtasks, i.e., for instance go to a station and pickup a passenger before heading to the passenger's destination. Finally, how to communicate and coordinate with each other, i.e., if taxi $T1$ is on its way to pick up a passenger at location *Blue*, taxi $T2$ should serve a passenger at one of the other stations. The state variables in this task are the locations of taxis (25 values each), status of taxis (5 values each, taxi is empty or transporting a passenger to one of the four stations), and status of stations B, G, R, and Y (4 values each, station is empty or has a passenger whose destination is one of the other three stations). Thus, in the multi-agent flat case, the size of the state space would grow to 4×10^6. The size of the Q table is this number multiplied by the number of primitive actions 10, which is 4×10^7. In the selfish multi-agent HRL algorithm, using state abstrac-

[4] Passenger arrival rate 10 indicates that on average, one passenger arrives at stations every 10 time steps.

tion and the fact that each agent stores only its own state variables, the number of the C and V values to be learned is reduced to $2 \times 135,895 = 271,790$, which is 135,895 values for each agent. In the *Cooperative HRL* algorithm, the number of values to be learned would be $2 \times 729,815 = 1,459,630$. Finally in the *COM-Cooperative HRL* algorithm, this number would be $2 \times 934,615 = 1,869,230$. In the *COM-Cooperative HRL*, we define *root* as a *cooperative subtask* and the highest level of the hierarchy as a *cooperation level* as shown in Figure 6.10. Thus, *root* is the only member of the *cooperation set* at that level, and $U_1 = A_{root} = \{GetB,\ GetG,\ GetR,\ GetY,\ Wait,\ Put\}$. The joint-action space for *root* is specified as the cross product of the *root* action set and U_1. Finally, $\tau_{continue}$ termination scheme is used for joint-action selection in this domain. All the experiments in this section were repeated five times and the results were averaged.

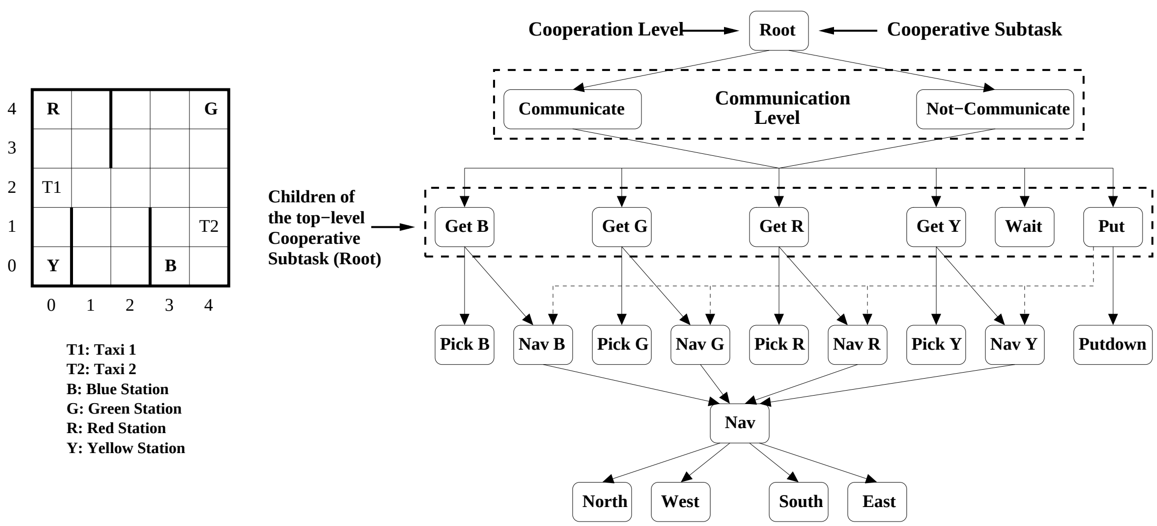


Figure 6.10. A multi-agent taxi domain and its associated task graph.

Figures 6.11 and 6.12 show the throughput of the system and the average waiting time per passenger for four algorithms, single-agent HRL, selfish multi-agent HRL, *Cooperative HRL*, and *COM-Cooperative HRL* when communication cost is zero.[5] As seen in Figures

[5]The *COM-Cooperative HRL* uses the task graph in Figure 6.10. The *Cooperative HRL* uses the same task graph without the *communication level*.

6.11 and 6.12, *Cooperative HRL* and *COM-Cooperative HRL* with $ComCost = 0$ have better throughput and average waiting time per passenger than selfish multi-agent HRL and single-agent HRL. The *COM-Cooperative HRL* learns slower than *Cooperative HRL*, due to more parameters to be learned in this model. However, it eventually converges to the same performance as the *Cooperative HRL* does.

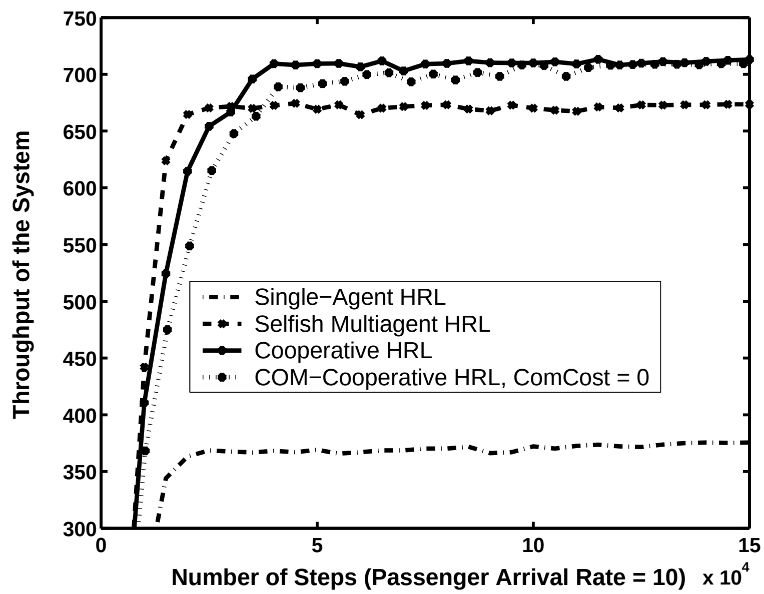


Figure 6.11. This figure shows that the *Cooperative HRL* and the *COM-Cooperative HRL* with $ComCost = 0$ have better throughput than the selfish multi-agent HRL and the single-agent HRL.

Figure 6.13 compares the average waiting time per passenger for the multi-agent selfish HRL and the *COM-Cooperative HRL* with $ComCost = 0$ for three different passenger arrival rates (5, 10, and 20). It demonstrates that as the passenger arrival rate becomes smaller, the coordination among taxis becomes more important. When taxis do not coordinate, it is possible that both taxis go to the same station. In this case, the first taxi picks up the passenger and the other one returns empty. This case can be avoided by incorporating coordination in the system. However, when the passenger arrival rate is high, there is a chance that a new passenger arrives after the first taxi picked up the previous passenger and

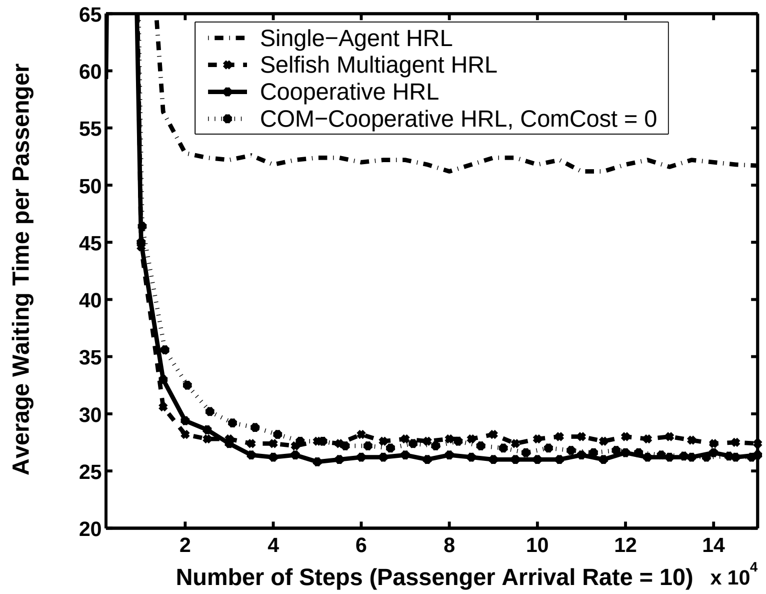


Figure 6.12. This figure shows that the average waiting time per passenger in the *Cooperative HRL* and the *COM-Cooperative HRL* with $ComCost = 0$ is less than the selfish multi-agent HRL and the single-agent HRL.

before the second taxi reaches the station. This passenger will be picked up by the second taxi. In this case, coordination would not be as crucial as the case when the passenger arrival rate is low.

Figure 6.14 demonstrates the relation between the communication policy and the communication cost. These two figures show the throughput and the average waiting time per passenger for the selfish multi-agent HRL and the *COM-Cooperative HRL* when the communication cost equals $0, 1, 5,$ and 10. In both figures, as the communication cost increases, the performance of the *COM-Cooperative HRL* becomes closer to the selfish multi-agent HRL. It indicates that when communication is expensive, agents learn not to communicate and to be selfish.

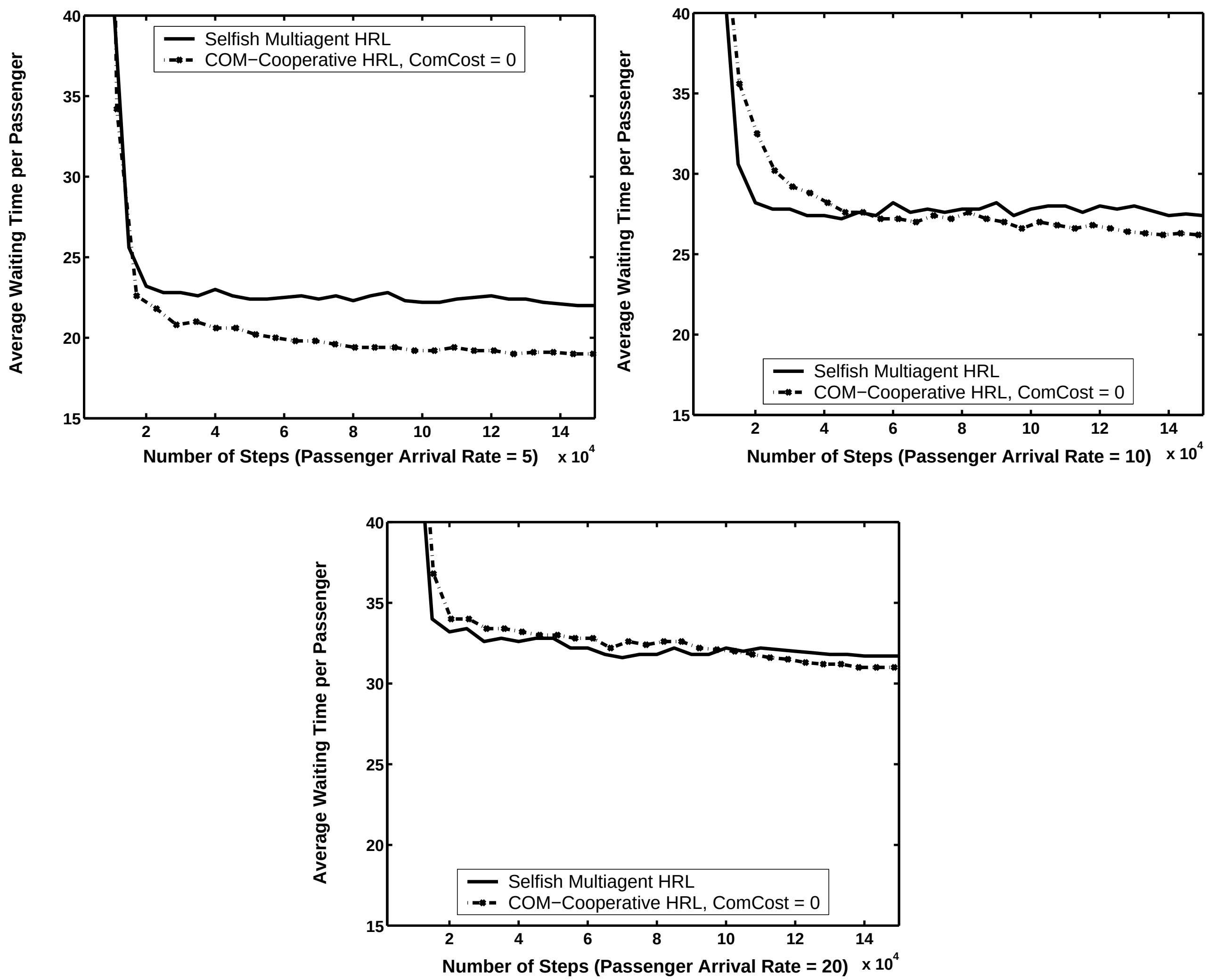


Figure 6.13. This figure compares the average waiting time per passenger for the selfish multi-agent HRL and the *COM-Cooperative HRL* with $ComCost = 0$ for three different passenger arrival rates (5, 10, and 20). It shows that coordination among taxis becomes more crucial as the passenger arrival rate becomes smaller.

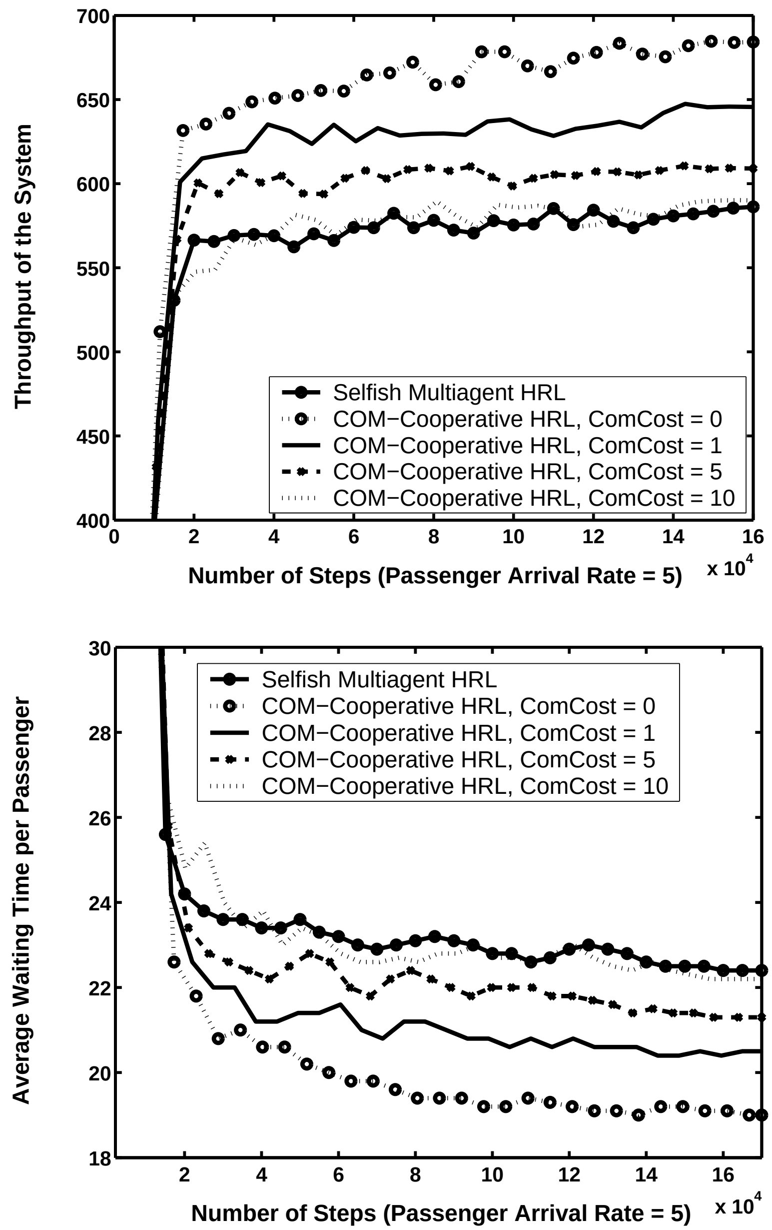


Figure 6.14. This figure shows that as communication cost increases, the throughput (top) and the average waiting time per passenger (bottom) of the *COM-Cooperative HRL* become closer to the selfish multi-agent HRL. It indicates that agents learn to be selfish when communication is expensive.

6.7 Summary and Future Work

In this chapter, we studied methods for learning to communicate and act in cooperative multi-agent systems using hierarchical reinforcement learning. The key idea underlying our approach is that coordination skills are learned much more efficiently if agents have a hierarchical representation of the task structure. The use of hierarchy speeds up learning in multi-agent domains by making it possible to learn coordination skills at the level of subtasks instead of primitive actions. A further advantage of this approach over flat learning methods is that, since high-level subtasks take a long time to complete, communication is needed fairly infrequently. We proposed two new cooperative multi-agent HRL algorithms, *Cooperative HRL* and *COM-Cooperative HRL* using the above idea. In both algorithms, agents are homogeneous, i.e., use the same task decomposition, learning is decentralized, and each agent learns three interrelated skills: how to perform subtasks, which order to do them in, and how to coordinate with other agents.

In *Cooperative HRL*, we assume communication is free and therefore agents do not need to decide if communication with their teammates is necessary. We demonstrate the efficacy of this algorithm using a four-agent AGV scheduling problem. We compare the performance of the *Cooperative HRL* algorithm with other algorithms such as selfish multi-agent HRL, single-agent HRL, and flat Q-learning in this domain. We also show that *Cooperative HRL* outperforms widely used industrial heuristics, such as *"first come first serve"*, *"highest queue first"*, and *"nearest station first"*.

In *COM-Cooperative HRL*, we address the issue of rational communicative behavior among autonomous agents. The goal is to learn both action and communication policies that together optimize the task given the communication cost. This algorithm is an extension of *Cooperative HRL* by including communication decisions in the model. We study the empirical performance of the *COM-Cooperative HRL* algorithm as well as the relation between the communication cost and the communication policy using a multi-agent taxi problem.

There are a number of directions for future work which can be briefly outlined. An immediate question that arises is to define the classes of cooperative multi-agent problems in which the proposed algorithms converge to a good approximation of optimal policy. The experiments of this chapter show that the effectiveness of these algorithms is most apparent in tasks where agents rarely interact at low levels (for example in the trash collection task, two robots may rarely need to exit through the same door at the same time). However, the algorithms can be generalized and adapted to constrained environments where agents are constantly running into one another (for example ten robots in a small room all trying to leave the room at the same time) by extending cooperation to lower levels of the hierarchy. This will result in a much larger set of action values that need to be learned, and consequently learning will be much slower, as shown in the AGV experiment depicted in Figure 6.8. A number of extensions would be useful, from studying the scenario where agents are heterogeneous, to recognizing the high-level subtasks being performed by other agents using a history of observations (plan recognition and activity modeling) instead of direct communication. In the later case, we assume that each agent can observe its teammates and uses its observations to extract their high-level subtasks. Good examples for this approach are games such as soccer, football or basketball, in which players often extract the strategy being performed by their teammates using recent observations instead of direct communication. Saria and Mahadevan (2004) presented a theoretical framework for online probabilistic plan recognition in cooperative multi-agent systems. Their model extends the abstract hidden Markov model (AHMM) (Bui et al., 2002) to cooperative multi-agent domains. We believe that the model presented by Saria and Mahadevan can be combined with the learning algorithms proposed in this chapter to reduce communication by learning to recognize the high-level subtasks being performed by the other agents.

Another direction for future work is to study different termination schemes for composing temporally extended actions. We used $\tau_{continue}$ termination strategy in the algorithms proposed in this chapter. However, it would be beneficial to investigate τ_{any} and τ_{all} termi-

nation schemes in our model. Many other manufacturing and robotics problems can benefit from these algorithms. Combining the proposed algorithms with function approximation and factored action models, which makes them more appropriate for continuous state problems, is also an important area of research. In this direction, we believe that the algorithms proposed in this chapter can be combined with the hierarchical policy gradient algorithms proposed in Chapter 5 to be used in multi-agent domains with continuous state and/or action. Finally, studying those communication features that have not been considered in our model such as message delay and probability of loss is another fundamental problem that needs to be addressed.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This dissertation demonstrates that by exploiting domain-specific properties, we can design more efficient hierarchical reinforcement learning (HRL) algorithms and scale up HRL to more complex large-scale problems. This chapter provides a summary of the methods and algorithms presented in this thesis, along with future questions that remain open.

7.1 Summary

In this dissertation, we investigated the use of hierarchy and abstraction as a means of solving complex sequential decision making problems, such as those with continuous state and/or continuous action spaces, and domains with multiple cooperative agents. We developed several novel extensions to HRL and designed algorithms that are appropriate for such problems.

Recent years have seen numerous successes of reinforcement learning (RL) approaches to control and decision making under uncertainty (Tesauro, 1994; Zhang and Dietterich, 1995; Singh and Bertsekas, 1996; Crites and Barto, 1998; Ng et al., 2004). However, the existing RL methods suffer from the curse of dimensionality: the exponential growth of the number of parameters to be learned with the size of any compact encoding of system state (Bellman, 1957). Recent attempts to combat the curse of dimensionality have turned to principled ways of exploiting abstraction in RL, which leads naturally to hierarchical control architectures and associated learning algorithms (Barto and Mahadevan, 2003). Although HRL approaches scale better than flat RL methods to high dimensional domains, they still suffer from the *curse of dimensionality*. Moreover, HRL methods have so far

only been studied in a narrow context: they have been investigated for the discrete-time discounted reward SMDP model; they have all been value function RL methods; and, they have only been studied in single-agent domains. The methods and algorithms developed in this dissertation expand the context and scope of HRL. They use prior knowledge in a principled way, and extend the existing HRL frameworks and algorithms to problems with continuous state and/or action spaces, and domains with multiple cooperative agents.

In Chapter 4, we presented new discrete-time and continuous-time *hierarchically optimal average reward RL* (HAR) and *recursively optimal average reward RL* (RAR) algorithms applicable to continuing tasks, including manufacturing, scheduling, queuing, and inventory control. These algorithms are based on the average-reward semi-Markov decision process (SMDP) model, which has been shown to be more appropriate for a wide class of continuing tasks than the better studied discounted reward SMDP model. The HAR algorithms aim to find a hierarchical policy within the space of policies defined by the hierarchical decomposition that maximizes the *global gain*. The RAR algorithms formulate subtasks in the hierarchy as continuing average reward problems, where the goal at each subtask is to maximize its gain given the policies of its children. We investigated the conditions under which the policy learned by the RAR algorithm at each subtask is independent of the context in which it is executed and therefore can be reused by other hierarchies. We demonstrated the performance of the proposed algorithms using two automated guided vehicle (AGV) scheduling tasks.

In Chapter 5, we described HPGRL, a family of *hierarchical policy gradient RL* algorithms for learning in domains with continuous state and/or continuous action spaces. We compared the performance of this family of algorithms with a hierarchical value function reinforcement learning (VFRL) algorithm and a flat RL algorithm in a simple taxi-fuel problem. The results demonstrated that the HPGRL algorithm converges slower than the hierarchical VFRL algorithm. To accelerate learning in HPGRL algorithms, we proposed a family of *hierarchical hybrid* algorithms in which subtasks located at high level(s) of the

hierarchy are formulated as VFRL, and subtasks located at low level(s) of the hierarchy are defined as policy gradient reinforcement learning (PGRL) problems. We used a continuous state and action ship steering task to illustrate this family of algorithms and to demonstrate their performance.

In Chapter 6, we studied methods for learning to communicate and act in cooperative multi-agent systems using hierarchical reinforcement learning. The key idea underlying our approach is that coordination skills are learned much more efficiently if agents have a hierarchical representation of the task structure. The use of hierarchy speeds up learning in multi-agent domains by making it possible to learn coordination skills at the level of subtasks instead of primitive actions. A further advantage of this approach over flat learning methods is that, since high-level subtasks take a long time to complete, communication is needed fairly infrequently. We proposed two new cooperative multi-agent HRL algorithms, *Cooperative HRL* and *COM-Cooperative HRL* using the above idea. In both algorithms, agents are homogeneous, i.e., use the same task decomposition, learning is decentralized and each agent learns three interrelated skills: how to perform subtasks, which order to do them in, and how to coordinate with other agents.

In *Cooperative HRL*, we assume communication is free and therefore agents do not need to decide if communication with their teammates is necessary. We demonstrated the efficacy of this algorithm using a four-agent AGV scheduling problem. We compared the performance of the *Cooperative HRL* algorithm with other algorithms such as selfish multi-agent HRL, single-agent HRL, and flat Q-learning in this domain. We also showed that *Cooperative HRL* outperforms widely used industrial heuristics, such as *"first come first serve"*, *"highest queue first"*, and *"nearest station first"*.

In *COM-Cooperative HRL*, we addressed the issue of rational communicative behavior among autonomous agents. The goal is to learn both action and communication policies that together optimize the task given the communication cost. This algorithm is an extension of *Cooperative HRL* by including communication decisions in the model. We studied

the empirical performance of the *COM-Cooperative HRL* algorithm as well as the relation between the communication cost and the communication policy using a multi-agent taxi problem.

7.2 Future Work

There are a number of directions for future work which are briefly outlined.

Hierarchical Average Reward Reinforcement Learning

An immediate question that arises is proving the asymptotic convergence of the algorithms proposed in Chapter 4 to hierarchically and recursively optimal average reward policies. These results should provide some theoretical validity to these algorithms, in addition to their empirical effectiveness demonstrated in Chapter 4. Studying other local optimality criteria for subtasks in a hierarchy is an interesting problem that needs to be addressed. It helps to develop more effective *recursively optimal average reward RL* algorithms. It is also obvious that many other manufacturing and robotics problems can benefit from the algorithms proposed in Chapter 4.

.

Hierarchical Policy Gradient Reinforcement Learning

The algorithms proposed in Chapter 5 are based on the assumption that the overall task (*root* of the hierarchy) is *episodic*. One direction for future work is to reformulate these algorithms for the case when the overall task is *continuing*. In this case, the *root* task is formulated as a *continuing* problem with the *average reward* as its performance function. Since the policy learned at *root* involves policies of its children, the type of optimality achieved at *root* depends on how we formulate other subtasks in the hierarchy. Different notions of optimality in *hierarchical average reward* presented in Chapter 4 can be used to develop new HPGRL algorithms for *continuing* problems.

Although the algorithms proposed in Chapter 5 give us the ability to deal with continuous state and/or continuous action spaces, they are not still appropriate for real-world

problems in which the speed of learning is crucial. The results of the ship steering task indicate that in order to apply the proposed algorithms to real-world domains, more powerful PGRL algorithms are needed — algorithms that need a smaller number of samples, and are less computationally expensive.

Hierarchical Multi-Agent Reinforcement Learning

An immediate question that arises is to define the classes of cooperative multi-agent problems in which the algorithms proposed in Chapter 6 converge to a good approximation of optimal policy. The experiments of this chapter show that the effectiveness of these algorithms is most apparent in tasks where agents rarely interact at low levels (for example in the trash collection task, two robots may rarely need to exit through the same door at the same time). However, the algorithms can be generalized and adapted to constrained environments where agents are constantly running into one another (for example ten robots in a small room all trying to leave the room at the same time) by extending cooperation to lower levels of the hierarchy. This will result in a much larger set of action values that need to be learned, and consequently learning will be much slower, as shown in the AGV experiment depicted in Figure 6.8. A number of extensions would be useful, from studying the scenario where agents are heterogeneous, to recognizing the high-level subtasks being performed by other agents using a history of observations (plan recognition and activity modeling) instead of direct communication. In the later case, we assume that each agent can observe its teammates and uses its observations to extract their high-level subtasks. Good examples for this approach are games such as soccer, football or basketball, in which players often extract the strategy being performed by their teammates using recent observations instead of direct communication. Saria and Mahadevan (2004) presented a theoretical framework for online probabilistic plan recognition in cooperative multi-agent systems. Their model extends the abstract hidden Markov model (AHMM) (Bui et al., 2002) to cooperative multi-agent domains. We believe that the model presented by Saria

and Mahadevan can be combined with the learning algorithms proposed in Chapter 6 to reduce communication by learning to recognize the high-level subtasks being performed by the other agents.

Another direction for future work in this area is to study different termination schemes for composing temporally extended actions. We used $\tau_{continue}$ termination strategy in the algorithms proposed in Chapter 6. However, it would be beneficial to investigate τ_{any} and τ_{all} termination schemes in our model. Many other manufacturing and robotics problems can benefit from these algorithms. Combining these algorithms with function approximation and factored action models, which makes them more appropriate for continuous state problems, is also an important area of research. In this direction, we believe that the algorithms proposed in Chapter 6 can be combined with the hierarchical policy gradient algorithms proposed in Chapter 5 to be used in multi-agent domains with continuous state and/or action. Finally, studying those communication features that have not been considered in our model such as message delay and probability of loss is another fundamental problem that needs to be addressed.

7.3 Closing Remarks

In this dissertation, we exploit domain-specific properties to design more efficient HRL algorithms. These algorithms extend HRL to solving complex sequential decision making problems such as those with continuous state and/or action spaces and domains with multiple cooperative agents. However, many issues remain to be studied before learning methods can be deployed in practical settings. In this chapter, we outlined a few open directions that are particularly related to the methods developed in this dissertation. Of course, there are many other more general open questions that must be addressed before effective learning techniques can be designed for tackling large-scale complex systems. Ultimately, we hope that such learning methods will aid human users in solving complex problems, which require learning and adaptation.

APPENDIX

INDEX OF SYMBOLS

Here we present a list of the symbols used in this dissertation to hopefully alleviate the difficulty for the reader, or at least provide a handy reference.

Notation	Definition
$\mathbb{R}$	set of real numbers
$\mathbb{N}$	set of natural numbers
E	expected value
$\mathcal{M}$	an MDP model
$\mathcal{S}$	set of states
$\mathcal{A}$	set of actions
$\mathcal{A}_s$	set of admissible actions in state s
$\mathcal{P}$	transition probability function in MDP and multi-step transition probability function in SMDP
$P(s'\|s,a)$	probability that action a causes transition from state s to state s' in an MDP
$\boldsymbol{P}^{\mu}$	transition probability matrix of policy μ in an MDP
$\bar{\boldsymbol{P}}^{\mu}$	limiting matrix of policy μ in an MDP
$\mathcal{R}$	reward function
$r(s,a)$	reward of taking action a in state s
I	initial state distribution
μ	a policy
$\mu(a\|s)$	probability that policy μ selects action a in state s
μ^*	optimal policy
γ	discount factor
α	learning rate parameter
V^{μ}	value function of policy μ in flat models
$V^{\boldsymbol{\mu}}$	hierarchical value function of hierarchical policy $\boldsymbol{\mu}$ in hierarchical models
$\hat{V}^{\boldsymbol{\mu}}$	projected value function of hierarchical policy $\boldsymbol{\mu}$ in hierarchical models
V^*	optimal value function
Q^{μ}	action-value function of policy μ in flat models
$Q^{\boldsymbol{\mu}}$	hierarchical action-value function of hierarchical policy $\boldsymbol{\mu}$ in hierarchical models
$\hat{Q}^{\boldsymbol{\mu}}$	projected action-value function of hierarchical policy $\boldsymbol{\mu}$ in hierarchical models
Q^*	optimal action-value function
Γ^*	Bellman operator

Notation	Definition
g^{μ}	average reward or gain of policy μ
$g^{\boldsymbol{\mu}}$	global gain under hierarchical policy $\boldsymbol{\mu}$
$g_i^{\boldsymbol{\mu}}$	local gain of subtask M_i under hierarchical policy $\boldsymbol{\mu}$
g^*	gain of optimal policy
H^{μ}	average-adjusted value function of policy μ in flat models
$H^{\boldsymbol{\mu}}$	hierarchical average-adjusted value function of hierarchical policy $\boldsymbol{\mu}$ in hierarchical models
$\hat{H}^{\boldsymbol{\mu}}$	projected average-adjusted value function of hierarchical policy $\boldsymbol{\mu}$ in hierarchical models
H^*	optimal average-adjusted value function
L^{μ}	average-adjusted action-value function of policy μ in flat models
$L^{\boldsymbol{\mu}}$	hierarchical average-adjusted action-value function of hierarchical policy $\boldsymbol{\mu}$ in hierarchical models
$\hat{L}^{\boldsymbol{\mu}}$	projected average-adjusted action-value function of hierarchical policy $\boldsymbol{\mu}$ in hierarchical models
L^*	optimal average-adjusted action-value function
$P(s', N\|s, a)$	probability that action a will cause the system to transition from state s to state s' in N time steps
$m(s'\|s, a)$	probability that an SMDP occupies state s' at the next decision epoch given that the agent takes action a in state s at the current decision epoch
$\boldsymbol{m}^{\mu}$	transition probability matrix of the embedded Markov chain of an SMDP for policy μ
$\bar{\boldsymbol{m}}^{\mu}$	limiting matrix of the embedded Markov chain of an SMDP for policy μ
$y(s, a)$	expected number of transition steps until the next decision epoch
$\mathcal{H}$	a hierarchy
M_i	subtask M_i in a hierarchy
S_i	set of states for subtask M_i in a hierarchy
$\|S_i\|$	cardinality of set of states S_i
A_i	set of actions for subtask M_i in a hierarchy
R_i	reward function for subtask M_i in a hierarchy
$\mathcal{I}_i$	initiation set for subtask M_i in a hierarchy
T_i	termination set for subtask M_i in a hierarchy
s_{T_i}	a terminal state of subtask M_i in a hierarchy ; $s_{T_i} \in T_i$
μ_i	a policy for subtask M_i in a hierarchy
$\boldsymbol{\mu}$	a hierarchical policy
$P_i^{\boldsymbol{\mu}}$	multi-step transition probability function of subtask M_i
$P_i^{\boldsymbol{\mu}}(s', N\|s)$	probability that action $\mu_i(s)$ causes transition from state s to state s' in N primitive steps under hierarchical policy $\boldsymbol{\mu}$
$F_i^{\boldsymbol{\mu}}$	multi-step abstract transition probability function of subtask M_i
$F_i^{\boldsymbol{\mu}}(s', N\|s)$	probability of transition from state s to state s' in N abstract actions taken by subtask M_i under hierarchical policy $\boldsymbol{\mu}$
$P^{\boldsymbol{\mu}}$	single-step transition probability function under hierarchical policy $\boldsymbol{\mu}$
$P^{\boldsymbol{\mu}}(s'\|s)$	probability that hierarchical policy $\boldsymbol{\mu}$ will cause the system to transition from state s to state s' at the level of primitive actions
$\boldsymbol{m}_i^{\boldsymbol{\mu}}$	transition probability matrix of the Markov chain at subtask M_i for hierarchical policy $\boldsymbol{\mu}$

Notation	Definition
$\bar{\boldsymbol{m}}_i^{\boldsymbol{\mu}}$	limiting matrix of the Markov chain at subtask M_i for hierarchical policy $\boldsymbol{\mu}$
Ω	set of possible values for Task-Stack in a hierarchy
$X = \Omega \times S$	joint state space of Task-Stack values and states in a hierarchy
$x = (\omega, s)$	joint state value x formed by Task-Stack value ω and state value s in a hierarchy
$\omega \nearrow i$	popping subtask M_i off Task-Stack with content ω in a hierarchy
$i \searrow \omega$	pushing subtask M_i onto Task-Stack with content ω in a hierarchy
$C^{\boldsymbol{\mu}}$	completion function of hierarchical policy $\boldsymbol{\mu}$
$CE^{\boldsymbol{\mu}}$	external completion function of hierarchical policy $\boldsymbol{\mu}$
π^{μ}	steady state probability vector of the Markov chain defined by policy μ
$\pi^{\mu}(s)$	steady state probability of being in state s for the Markov chain defined by policy μ
$\boldsymbol{\theta}$	set of policy parameters
$\boldsymbol{\theta}_i$	set of policy parameters for subtask M_i
$\mu_i(\boldsymbol{\theta}_i)$	policy for subtask M_i corresponding to parameter vector $\boldsymbol{\theta}_i$
$\boldsymbol{\mu}(\boldsymbol{\theta})$	hierarchical policy corresponding to parameter vector $\boldsymbol{\theta}$
$\chi_i(\boldsymbol{\theta})$	weighted reward-to-go of subtask M_i under hierarchical policy parameterized by parameter set $\boldsymbol{\theta}$
$J_i(s; \boldsymbol{\theta})$	reward-to-go of subtask M_i in state s under hierarchical policy parameterized by parameter set $\boldsymbol{\theta}$
Υ	set of a finite collection of agents in multi-agent SMDP

BIBLIOGRAPHY

Abounadi, J., Bertsekas, D. P., and Borkar, V. S. (2001). Learning algorithms for Markov decision processes with average cost. *SIAM Journal on Control and Optimization*, 40:681–698.

Andre, D. (2003). *Programmable Reinforcement Learning Agents*. PhD thesis, University of California at Berkeley.

Andre, D. and Russell, S. J. (2001). Programmable reinforcement learning agents. In *Proceedings of Advances in Neural Information Processing Systems 13*, pages 1019–1025. MIT Press.

Andre, D. and Russell, S. J. (2002). State abstraction for programmable reinforcement learning agents. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 119–125.

Askin, R. and Standridge, C. (1993). *Modeling and Analysis of Manufacturing Systems*. John Wiley and Sons.

Balch, T. and Arkin, R. (1998). Behavior-based formation control for multi-robot teams. *IEEE Transactions on Robotics and Automation*, 14:1–15.

Barto, A. and Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Systems (Special Issue on Reinforcement Learning)*, 13:41–77.

Baxter, J. and Bartlett, P. (2001). Infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:319–350.

Baxter, J., Bartlett, P., and Weaver, L. (2001). Experiments with infinite-horizon policy-gradient estimation. *Journal of Artificial Intelligence Research*, 15:351–381.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press.

Bernstein, D., Zilberstein, S., and Immerman, N. (2000). The complexity of decentralized control of Markov decision processes. In *Proceedings of the Sixteenth International Conference on Uncertainty in Artificial Intelligence*, pages 32–37.

Bertsekas, D. (1995). *Dynamic Programming and Optimal Control*. Athena Scientific.

Bertsekas, D. (1998). A new value iteration method for the average cost dynamic programming problem. *SIAM on Control and Optimization*, 36:742–759.

Bertsekas, D. and Tsitsiklis, J. (1996). *Neuro-Dynamic Programming*. Athena Scientific.

Blackwell, D. (1962). Discrete dynamic programming. *Ann. Math. Stat.*, 33:719–726.

Boutilier, C. (1999). Sequential optimality and coordination in multiagent systems. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 478–485.

Bowling, M. and Veloso, M. (2002). Multiagent learning using a variable learning rate. *Artificial Intelligence*, 136:215–250.

Bradtke, S. and Duff, M. (1995). Reinforcement learning methods for continuous-time Markov decision problems. In *Proceedings of Advances in Neural Information Processing Systems 7*, pages 393–400. MIT Press.

Brafman, R. and Tennenholtz, M. (1997). Modeling agents as qualitative decision makers. *Artificial Intelligence*, 94:217–268.

Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, pages 14–23.

Bui, H., Venkatesh, S., and West, G. (2002). Policy recognition in the abstract hidden Markov model. *Journal of Artificial Intelligence Research*, 17:451–499.

Cao, X., Ren, Z., Bhatnagar, S., Fu, M., and Marcus, S. (2002). A time aggregation approach to Markov decision processes. *Automatica*, 38:929–943.

Cassandras, C. and Lafortune, S. (1999). *Introduction to Discrete Event Systems*. Kluwer Academic Publishers.

Crites, R. and Barto, A. (1998). Elevator group control using multiple reinforcement learning agents. *Machine Learning*, 33:235–262.

Currie, K. and Tate, A. (1991). O-plan: The open planning architecture. *Artificial Intelligence*, 52(1):1104–1111.

Dayan, P. and Hinton, G. (1993). Feudal reinforcement learning. In *Proceedings of Advances in Neural Information Processing Systems 5*, pages 271–278.

de Farias, D. P. (2002). *The Linear Programming Approach to Approximate Dynamic Programming: Theory and Application*. PhD thesis, Stanford University.

Dietterich, T. (1998). The MAXQ method for hierarchical reinforcement learning. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 118–126.

Dietterich, T. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303.

Dietterich, T. and Wang, X. (2002). Batch value function approximation via support vectors. In *Proceedings of Advances in Neural Information Processing Systems 14*, pages 1491–1498.

Digney, B. (1996). Emergent hierarchical control structures: Learning hierarchical/reactive relationships in reinforcement learning environments. In *From Animals to Animats 4*, pages 363–373.

Drescher, G. (1991). *Made-up Minds, A Constructivist Approach to Artificial Intelligence*. MIT Press.

Filar, J. and Vrieze, K. (1997). *Competitive Markov Decision Processes*. Springer Verlag.

Forestier, J. and Varaiya, P. (1978). Multilayer control of large Markov chains. *IEEE Transactions on Automatic Control*, 23(2):298–304.

Gershwin, S. (1994). *Manufacturing Systems Engineering*. Prentice Hall.

Ghavamzadeh, M. and Mahadevan, S. (2001). Continuous-time hierarchical reinforcement learning. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 186–193.

Ghavamzadeh, M. and Mahadevan, S. (2002). Hierarchically optimal average reward reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 195–202.

Ghavamzadeh, M. and Mahadevan, S. (2003). Hierarchical policy gradient algorithms. In *Proceedings of the Twentieth International Conference on Machine Learning*, pages 226–233.

Ghavamzadeh, M. and Mahadevan, S. (2004). Learning to communicate and act using hierarchical reinforcement learning. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1114–1121.

Gordon, G. (1999). *Approximate Solutions to Markov Decision Processes*. PhD thesis, Carnegie Mellon University.

Guestrin, C., Koller, D., and Parr, R. (2001). Max-norm projections for factored MDPs. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 673–680.

Guestrin, C., Lagoudakis, M., and Parr, R. (2002). Coordinated reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 227–234.

Hengst, B. (2002). Discovering hierarchy in reinforcement learning with HEXQ. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 243–250.

Ho, Y. and Cao, X. (1991). *Perturbation Analysis of Discrete Event Dynamic Systems*. Kluwer.

Howard, R. (1960). *Dynamic Programming and Markov Processes*. MIT Press.

Howard, R. (1971). *Dynamic Probabilistic Systems: Semi-Markov and Decision Processes*. John Wiley and Sons.

Hu, J. and Wellman, M. (1998). Multiagent reinforcement learning: Theoretical framework and an algorithm. In *Proceedings of the Fifteenth International Conference on Machine Learning*, pages 242–250.

Huber, M. and Grupen, R. (1997). A feedback control structure for online learning tasks. *Robotics and Autonomous Systems*, 22:303–315.

Jaakkola, T., Jordan, M., and Singh, S. (1994). On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6(6):1185–1201.

Jonsson, A. and Barto, A. (2005). A causal approach to hierarchical decomposition of factored MDPs. In *Proceedings of the Twenty Second International Conference on Machine Learning*.

Kaelbling, L. (1993a). Hierarchical reinforcement learning: Preliminary results. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 167–173.

Kaelbling, L. (1993b). Learning to achieve goals. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 1094–1098.

Kaelbling, L., Littman, M., and Moore, A. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285.

Kearns, M., Littman, M., and Singh, S. (2001). Graphical models for game theory. In *Proceedings of the Thirteenth International Conference on Uncertainty in Artificial Intelligence*, pages 253–260.

Kearns, M., Mansour, Y., and Ng, A. (2000). Approximate planning in large POMDPs via reusable trajectories. In *Proceedings of Advances in Neural Information Processing Systems 12*, pages 1001–1007. MIT Press.

Kimura, H., Yamamura, M., and Kobayashi, S. (1995). Reinforcement learning by stochastic hill-climbing on discounted reward. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 295–303.

Klein, C. and Kim, J. (1996). AGV dispatching. *International Journal of Production Research*, 34:95–110.

Knoblock, C. (1990). Learning abstraction hierarchies for problem solving. In *Proceedings of the Eight National Conference on Artificial Intelligence*, pages 923–928.

Kokotovic, P., Khalil, H., and O'Reilly, J. (1986). *Singular Perturbation Methods in Control: Analysis and Design*. Academic Press.

Koller, D. and Milch, B. (2001). Multiagent influence diagrams for representing and solving games. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 1027–1034.

Koller, D. and Parr, R. (2000). Policy iteration for factored MDPs. In *Proceedings of the Sixteenth International Conference on Uncertainty in Artificial Intelligence*, pages 326–334.

Konda, V. (2002). *Actor-Critic Algorithms*. PhD thesis, Massachusetts Institute of Technology.

Korf, R. (1985). Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77.

La-Mura, P. (2000). Game networks. In *Proceedings of the Sixteenth International Conference on Uncertainty in Artificial Intelligence*.

Laird, J., Rosenbloom, P., and Newell, A. (1986). Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46.

Lesser, V., Ortiz, C., and Tambe, M. (2003). *Distributed Sensor Networks: A Multiagent Perspective*. Kluwer Academic Publishers.

Lin, L. (1993). *Reinforcement Learning for Robots using Neural Networks*. PhD thesis, Carnegie Mellon University.

Littman, M. (1994). Markov games as a framework for multiagent reinforcement learning. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 157–163.

Littman, M. (2001). Friend-or-foe Q-learning in general-sum games. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 322–328.

Littman, M., Kearns, M., and Singh, S. (2002). An efficient exact algorithm for singly connected graphical games. In *Proceedings of Advances in Neural Information Processing Systems 14*, pages 817–824. MIT Press.

Mahadevan, S. (1996). Average reward reinforcement learning: foundations, algorithms, and empirical results. *Machine Learning*, 22:159–196.

Mahadevan, S. and Connell, J. (1992). Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55:311–365.

Mahadevan, S., Khaleeli, N., and Marchalleck, N. (1997a). Designing agent controllers using discrete-event Markov models. In *Proceedings of the AAAI Fall Symposium on Model-Directed Autonomous Systems*.

Mahadevan, S., Marchalleck, N., Das, T., and Gosavi, A. (1997b). Self-improving factory simulation using continuous-time average reward reinforcement learning. In *Proceedings of the Fourteenth International Conference on Machine Learning*, pages 182–190.

Mannor, S., Menache, I., Hoze, A., and Klein, U. (2004). Dynamic abstraction in reinforcement learning via clustering. In *Proceedings of the Twenty-First International Conference on Machine Learning*, pages 560–567.

Marbach, P. (1998). *Simulated-Based Methods for Markov Decision Processes*. PhD thesis, Massachusetts Institute of Technology.

Mataric, M. (1997). Reinforcement learning in the multi-robot domain. *Autonomous Robots*, 4:73–83.

McGovern, A. and Barto, A. (2001). Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the Eighteenth International Conference on Machine Learning*, pages 361–368.

Mealeau, N., Peshkin, L., Kim, K.-E., and Kaelbling, L. (1999). Learning finite-state controllers for partially observable environments. In *Proceedings of the Fifteenth International Conference on Uncertainty in Artificial Intelligence*, pages 427–436.

Menache, I., Mannor, S., and Shimkin, N. (2002). Q-cut dynamic discovery of subgoals in reinforcement learning. In *Proceedings of the Thirteenth European Conference on Machine Learning*, pages 295–306.

Miller, W., Sutton, R., and Werbos, P. (1990). *Neural Networks for Control*. MIT Press.

Moore, A. and Atkeson, C. (1993). Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130.

Morimoto, J. and Doya, K. (2001). Acquisition of stand-up behavior by a real robot using hierarchical reinforcement learning. *Robotics and Autonomous Systems*, 36:37–51.

Ng, A. (2003). *Shaping and Policy Search in Reinforcement Learning*. PhD thesis, University of California at Berkeley.

Ng, A., Harada, D., and Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 278–287.

Ng, A. and Jordan, M. (2000). Pegasus: A policy search method for large MDPs and POMDPs. In *Proceedings of the Sixteenth International Conference on Uncertainty in Artificial Intelligence*, pages 406–415.

Ng, A., Kim, H., Jordan, M., and Sastry, S. (2004). Autonomous helicopter flight via reinforcement learning. In *Proceedings of Advances in Neural Information Processing Systems 16*. MIT Press.

Oates, T. and Cohen, P. (1996). Searching for planning operators with context dependent and probabilistic effects. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 863–868.

Ortiz, L. and Kearns, M. (2003). Nash propagation for loopy graphical games. In *Proceedings of Advances in Neural Information Processing Systems 15*. MIT Press.

Owen, G. (1995). *Game Theory*. Academic Press.

Parr, R. (1998). *Hierarchical Control and Learning for Markov Decision Processes*. PhD thesis, University of California at Berkeley.

Peshkin, L., Kim, K., Meuleau, M., and Kaelbling, L. (2000). Learning to cooperate via policy search. In *Proceedings of the Sixteenth International Conference on Uncertainty in Artificial Intelligence*, pages 489–496.

Pickett, M. and Barto, A. (2002). Policyblocks: An algorithm for creating useful macro-actions in reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 506–513.

Precup, D. (2000). *Temporal Abstraction in Reinforcement Learning*. PhD thesis, University of Massachusetts Amherst.

Puterman, M. (1994). *Markov Decision Processes*. Wiley Interscience.

Pynadath, D. and Tambe, M. (2002). The communicative multiagent team decision problem: Analyzing teamwork theories and models. *Journal of Artificial Intelligence Research*, 16:389–426.

Rohanimanesh, K. and Mahadevan, S. (2003). Learning to take concurrent actions. In *Proceedings of Advances in Neural Information Processing Systems 15*. MIT Press.

Rummery, G. and Niranjan, M. (1994). *On-line Q-learning using Connectionist Systems*. Technical Report CUED/F-INFENG/TR 166, Engineering Department, Cambridge University.

Sacerdoti, E. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2):115–135.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3:210–229.

Saria, S. and Mahadevan, S. (2004). Probabilistic plan recognition in multiagent systems. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling*, pages 12–22.

Schneider, J., Wong, W., Moore, A., and Riedmiller, M. (1999). Distributed value functions. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 371–378.

Schwartz, A. (1993). A reinforcement learning method for maximizing undiscounted rewards. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 298–305.

Seri, S. and Tadepalli, P. (2002). Model-based hierarchical average-reward reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pages 562–569.

Simon, H. A. (1981). *The Sciences of the Artificial*. MIT Press. second edition.

Simsek, O. and Barto, A. (2004). Using relative novelty to identify useful temporal abstractions in reinforcement learning. In *Proceedings of the Twenty-First International Conference on Machine Learning*, pages 751–758.

Singh, S. (1992). Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8:323–339.

Singh, S. and Bertsekas, D. (1996). Reinforcement learning for dynamic channel allocation in cellular telephone systems. In *Proceedings of Advances in Neural Information Processing Systems 9*, pages 974–980.

Singh, S., Jaakkola, T., Littman, M., and Szepesvari, C. (2000a). Convergence results for single-step on-policy reinforcement learning algorithms. *Machine Learning*, 38(3):287–308.

Singh, S., Kearns, M., and Mansour, Y. (2000b). Nash convergence of gradient dynamics in general-sum games. In *Proceedings of the Sixteenth International Conference on Uncertainty in Artificial Intelligence*, pages 541–548.

Stone, P. and Veloso, M. (1999). Team-partitioned, opaque-transition reinforcement learning. In *Proceedings of the Third International Conference on Autonomous Agents*, pages 206–212.

Sugawara, T. and Lesser, V. (1998). Learning to improve coordinated actions in cooperative distributed problem-solving environments. *Machine Learning*, 33:129–154.

Sutton, R. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44.

Sutton, R. (1991). Dyna, an integrated architecture for learning, planning, and reacting. *SIGART Bulletin*, 2:160–163.

Sutton, R. and Barto, A. (1998). *An Introduction to Reinforcement Learning*. MIT Press.

Sutton, R., Precup, D., and Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211.

Tadepalli, P. and Ok, D. (1996). Auto-exploratory average reward reinforcement learning. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 881–887.

Tadepalli, P. and Ok, D. (1998). Model-based average reward reinforcement learning. *Artificial Intelligence*, 100:177–224.

Tan, M. (1993). Multiagent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the Tenth International Conference on Machine Learning*, pages 330–337.

Tesauro, G. (1994). TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219.

Thrun, S. and Schwartz, A. (1995). Finding structure in reinforcement learning. In *Proceedings of Advances in Neural Information Processing Systems 8*, pages 385–392.

Van-Roy, B. (1998). *Learning and Value Function Approximation in Complex Decision Processes*. PhD thesis, Massachusetts Institute of Technology.

Vickrey, D. and Koller, D. (2002). Multiagent algorithms for solving graphical games. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 345–351.

Wang, G. and Mahadevan, S. (1999). Hierarchical optimization of policy-coupled semi-Markov decision processes. In *Proceedings of the Sixteenth International Conference on Machine Learning*, pages 464–473.

Watkins, C. (1989). *Learning from Delayed Rewards*. PhD thesis, Kings College, Cambridge, England.

Weiss, G. (1999). *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press.

Williams, J. and Singh, S. (1999). Experiments with an algorithm which learns stochastic memoryless policies for POMDPs. In *Proceedings of Advances in Neural Information Processing Systems 11*, pages 1073–1079.

Williams, R. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256.

Xuan, P. and Lesser, V. (2002). Multiagent policies: from centralized ones to decentralized ones. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1098–1105.

Xuan, P., Lesser, V., and Zilberstein, S. (2001). Communication decisions in multiagent cooperation: Model and experiments. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 616–623.

Zhang, W. and Dietterich, T. (1995). A reinforcement learning approach to job-shop scheduling. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1114–1120.